RETROSPECTIVE:

A Data Locality Optimizing Algorithm

Monica S. Lam
Computer Systems Laboratory
Stanford University

1. INTRODUCTION

"A Data Locality Optimizing Algorithm" was one of the first papers published as part of the SUIF parallelizing compiler research project, which lasted from 1989 to 2001. The main research objective of the SUIF project was to improve data locality for uniprocessors and to maximize parallelism and minimize communication for multiprocessors. We focused on two approaches: loop transformations on kernels and interprocedural analysis to find coarse-grain parallelism. SUIF stands for the Stanford University Intermediate Format, and the compiler infrastructure developed has been made publicly available and used in many compiler research projects.

This paper presented a loop transformation algorithm to enhance data locality for uniprocessors and multiprocessors. I would not have guessed in 1991 when this paper was published that it would take ten years before we arrived at a solution that I felt satisfactorily solved the complete problem. I would like to take this opportunity to describe our discoveries along this journey.

Michael Wolf and I started our research with the goal of improving data locality on uniprocessor machines by automating blocking. We found ourselves solving the parallelization problem before we came up with our data locality algorithm. Both our algorithms for parallelism and data locality use unimodular transforms, which we developed to unify loop interchange, reversal and skewing. Our unimodular transformation algorithms are applicable only to perfectly nested loops and are built upon the imprecise abstraction of direction vectors. Jennifer Anderson and I studied the problem of automatically decomposing computation and data across multiprocessors, focusing on the minimization of communication across sequences of loops. Amy Lim and I revisited data locality and parallelization with the goal of handling general loop structures. We generalized unimodular transforms to affine partitioning, which unifies in addition the techniques of fusion, fission, reindexing, scaling and statement reordering. Operating directly on access functions in the code instead of direction vectors, our parallelization algorithm is provably optimal in maximizing parallelism and minimizing communication. This approach also generalizes blocking and array contraction, two important optimizations for locality.

I wish I could claim that I have carefully broken down this challenging problem into a ten-year research plan. Instead, we had simply picked pressing research problems at the time and provided the best solution we could at each step. We discovered the structure of the problem part by part as we solved each subproblem. In fact, I think that the success of this research rested partly on the fact that each step along the way addressed some real-life problem. This helped us focus on important issues and ensured that the basic approach was correct.

2. SYSTOLIC ALGORITHMS

Our approach to loop transformations was inspired by systolic algorithms, which were intended to be implemented directly in silicon. Over a hundred papers have been written on systolic algorithms showing that many numerical algorithms can be mapped onto regular arrays of processing units communicating in a simple and regular manner. Most of these numerical algorithms are trivially parallelizable and the challenge in systolic array design is in minimizing communication between processors, which of course is also the key to effective parallelism on multiprocessors.

Not only did systolic array research teach us what efficient parallel algorithms look like, the design methodology that emerged provided important insights to automatic parallelization. It was shown that a systolic computation can be represented as an index set and its data dependences can be represented as distance vectors between points in the index set. Systolic array designs can be captured by geometric transforms that project the index sets representing the computations onto axes representing time and space. Dependences must point forwards in time for the design to be valid, and cross-processor dependences represent communication. This geometric model forms the basis of our work on unimodular transforms.

3. LOCALITY AND PARALLELISM

What has data locality got to do with parallelism? In each of my attempts to optimize data locality, first with Michael Wolf then with Amy Lim, we ended up finding an algorithm for parallelization first before solving the locality problem. We did not realize that parallelism is a special case of locality optimization.

Consider the special case where a computation is made up of totally independent threads of operations that share no common data, and that the target machine has only single-word data cache lines. Since all data reuse happens within a single thread, clearly executing these threads sequentially would maximize the advantage of data reuse. That is, to maximize data locality, we must first tease apart all the independent threads of computation, which is precisely the basic parallelization problem. Data locality optimization is more complicated as it must also consider the sharing of common input data and multiple-word cache lines.

Moreover, if there is no parallelism in the program, all operations must be sequentially ordered. Thus, parallelization analysis is also useful for locality optimization in that it identifies the opportunities for code transformations.

4. FROM UNIMODULAR TO AFFINE

The following outlines the research that led us from unimodular transforms to affine partitioning, highlighting the important lessons we learned along the way. All the techniques described here are applicable only to codes that access arrays using indices that are affine functions of enclosing loop indices.

4.1 Unimodular Transforms

Blocking, also known as tiling, was a hot topic in 1991. Fast on-chip floating-point units had just started to appear in general-purpose microprocessors, making them attractive as computation engines for numerical applications. However, unlike vector machines whose memory subsystems are designed to support scientific computing, microprocessors rely on caches which have been designed for integer applications. Blocked algorithms, originally developed by numerical analysts to minimize disk accesses, are found to be effective for machines with caches. Thus, blocking helps establish microprocessors as a lower cost alternative to vector machines.

Most of the work on loop transformations at the time was based on pairwise source-to-source transformations developed for vectorization. Influenced by systolic arrays, Michael Wolf and I modeled transformations of perfect loop nests as unimodular transforms on the iteration space, whereby unifying the transforms of loop interchange, reversal and skewing[9]. The domain of systolic algorithms was limited to computations whose dependences can be represented as distance vectors, and can be executed in $O(n)$ time where n represents the number of iterations in each loop. We generalized the domain to include all perfect loop nests, augmenting distance vectors with direction vectors from the vectorization literature. In this model, time may be multi-dimensional: a transform is considered legal if and only if the transformed dependences are lexicographically positive.

Both systolic synthesis and vectorization represent data dependences by the differences between related iterations, and optimize the code according to the principle that dependence vectors must not cross iteration boundaries in a parallel loop. Direction vectors, while grossly inaccurate, are adequate for unimodular transforms for perfect loop nests. They are too imprecise, however, for the more general transformation of affine partitioning on arbitrary loop nests.

The most important and lasting contribution this work made was the concept of a canonical form for parallelism. In this representation, each fully permutable subnest is made as large as possible, starting with the outermost nest. (A sequential loop is trivially fully permutable.) Each such subnest can be wavefronted, blocked, and executed in at most $O(n)$ time, where n is the number of iterations in one loop. This hierarchical representation was instrumental to the development of our parallelization algorithm based on affine partitioning.

Our paper "A Data Locality Optimizing Algorithm" presented an automatic blocking algorithm for perfect loop nests on uniprocessors and multiprocessors[8]. The paper's most important contributions are (1) a mathematical model for evaluating data reuse in affine data access functions and (2) the basic principle that blocking exploits reuse in multiple dimensions. Experimental results suggest that our proposed heuristics work quite well as long as the loops are perfectly nested. Unfortunately, many codes require optimization across loop nests.

The industry at the time was quite eager to adopt results from compiler research that improved locality. The fact that a matrix multiplication routine was included in spec92, a popular benchmark used to evaluate processor performance, probably helped raise the industry's interest on blocking. Michael Wolf joined SGI and implemented a version of unimodular transforms and blocking in the SGI compiler. Many other commercial compilers implemented some form of a blocking algorithm.

4.2 Data and Computation Decompositions

In the early part of the 1990s, many multiprocessor architectures, with shared or distributed address spaces, were commercially available. Examples of distributed memory machines include the Cray T3E, the IBM SP2, the Intel Paragon and Thinking Machine's CM-5; examples of shared memory machines include the Digital AlphaServer and the Silicon Graphics Power Challenge. Minimizing communication between processors is critical to developing efficient programs for these machines. Many companies and universities worked together to create High-Performance Fortran (HPF), a Fortran-90 dialect augmented with user-specified data decompositions. HPF compilers must automatically determine, based on the data decompositions, how the computation should be decomposed across the processors and generate the necessary communication code.

Jennifer Anderson and I recognized that data and computation decompositions are inherently tied together and should be solved at the same time. By the time users can specify the data decompositions properly, they must have already figured out where the parallelism is. Furthermore, it may be simpler to derive both data and computation decompositions automatically instead of having to handle all the data decompositions that can be specified.

Our algorithm[1] to find data and computation decompositions has two steps. It first applies unimodular transforms to find all the parallelization opportunities in each loop nest. Second, it chooses between the available ways to parallelize the loops so as to minimize communication. This two-step approach makes the algorithm simpler but suboptimal. This work's main contributions are (1) the basic communication minimization principle that "an iteration and the data it uses should be assigned to the same processor to avoid communication", and (2) heuristics that operate on data access functions in the code directly than direction vectors.

4.3 Affine Partitioning for Parallelism

In 1994, Amy Lim and I started working on an integrated algorithm that improves parallelism and minimizes communication of arbitrary loop nests. There were many outstanding questions to answer: How should we generalize unimodular transforms to arbitrary loop nesting structures? What is the analog of blocking, which is defined only for perfect loop nests?

Our solution to the first question is to represent the structure of an arbitrary program faithfully; it is simply not possible to find the best transform otherwise. A simple statement is treated as a loop with a single iteration. Thus, every dynamic operation is uniquely identified by the values of the indices of the enclosing loops. Loop indices can be organized as a tree reflecting the block structure of the program. A transformed program has a new tree of indices. In affine partitioning, an affine transform is created for each operation, mapping its old index values to new index values. This model encompasses all the possible combinations of unimodular transforms (interchange, skewing and reversal), fusion, fission, reindexing, scaling and statement reordering.

Our technique does not use direction vectors, but operates directly on affine data access functions. To find parallelism without synchronization, we require that "if two operations use the same data, they should be mapped to the same processor". By eliminating the mention of data mappings from the principle Jennifer and I used, this constraint allows arbitrary data mappings and is thus more powerful. Using the Farkas lemma[7], which we learned from studying Feautrier's work[2], we showed that we can reduce the problem of finding synchronization-free parallelism to finding all independent solutions to a set of linear constraints[3].

We solved synchronization-free parallelization quickly, but took much longer to figure out how to find parallelism that requires synchronization. From the work on unimodular transforms, we knew that it is useful to find largest outermost fully permutable loop nests. Data dependence considerations dictate that "if two operations from a sequential program use the same data, their relative ordering must be preserved". Our key discovery was to realize that if there are multiple schedules that can satisfy the data dependence constraints, then there must be flexibility in scheduling, which in turn means that the code is parallelizable. The beauty is that we can find the largest fully permutable loop nest in a manner similar to how we find the largest fully parallel loop nest: that is, by finding all the independent solutions to a set of linear constraints. The difference is that for permutability, we seek a mapping of the computation to the time axes; whereas for parallelism, we seek a mapping of the computation to the space (processor) axes.

To find all the parallelism in a program, the two basic steps are repeated recursively to create the same canonical program representation used in unimodular transforms. The difference is that we introduced a small step to break apart the strongly connected components in the body of each fully permutable subnest. Our algorithm is proven to maximize the degree of parallelism while minimizing the degree of synchronizations[4, 5].

4.4 Affine Partitioning for Locality

Almost exactly ten years after Michael Wolf and I published the data locality algorithm for perfect loop nests, Amy Lim, Shih-wei Liao and I published an algorithm for optimizing locality across arbitrary loop nests based on affine partitioning[6]. The algorithm first puts the code in the canonical form for parallelization. This code minimizes synchronization but may not have good data locality. Next, the algorithm analyzes data reuse in the program using the techniques in the original locality optimization paper. To improve locality, strongly connected components sharing reuse are fused and parallel threads sharing reuse are interleaved. The latter is achieved by a generalization of blocking for imperfect loops. Instead of "stripmine and interchange", blocking now means "stripmine and distribute". Here, outer parallel loops with reuse are stripmined and moved into the innermost loops. Moreover, by grouping all the related instructions together, this approach exposes many more opportunities for array contraction than ever possible before.

From our experiments, we see that this algorithm works well for codes as long as their array access functions are affine. The transformed code often looks significantly different from the original code and can be much more efficient than code written by a reasonably experienced application developer.

5. CONCLUDING REMARKS

This work builds upon years of research on loop transformations for vectorization and parallelization, most notably by research groups led by David Kuck at University of Illinois at Urbana-Champaign, Ken Kennedy at Rice University, Fran Allen at IBM research, Paul Feautrier, and Francois Irigoin in Ecole des Mines. We have come a long way starting with the pioneering work that discovered the many equivalent ways of expressing the same computation, interactive systems that help programmers apply transforms, and to the invention of program abstractions and automatic transformation techniques.

Programs whose data access functions are affine are special in that dynamically executed operations have very simple relationships with the data elements they access. Such information is fully exploited by array partitioning. Affine partitioning solves the fundamental scheduling constraints, as imposed by a program's data dependences and the goal to eliminate communication, without introducing any imprecise abstractions. Affine partitioning appears also to be sufficiently general and powerful, as it covers most of the loop transformations discovered through years of parallelization and vectorization research.

6. ACKNOWLEDGMENT

I would like to thank John Hennessy for his help through the years; I cannot imagine a better research environment to have started my career than the one he provided. I also want to thank the many members of the SUIF group for their dedication to the project: Gerald Aigner, Saman Amarasinghe, Jennifer Anderson, Gerald Cheong, Amer Diwan, Robert French, Anwar Ghuloum, Mary Hall, David Heine, Shih-wei Liao, Amy Lim, Jing Yee Lim, Vladimir Livshits, Dror Maydan, Todd Mowry, Brian Murphy, Karen Pieper, Patrick Sathyanathan, Mike Smith, Steven Tjiang, Chau-Wen Tseng, Robert Wilson, Christopher Wilson and Michael Wolf. The funding of the SUIF project was provided by DARPA, NSF and the Department of Energy. I wish to thank my DARPA program managers, Gil Weigand, Bob Lucas and Frederica Darema, for their support of this project.

REFERENCES

[1] J. M. Anderson and M. S. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 112–125, June 1993.

[2] P. Feautrier. Dataflow analysis of scalar and array references. *Journal of Parallel and Distributed Computing*, 20(1):23–53, February 1991.

[3] A. W. Lim and M. S. Lam. Communication-free parallelization via affine transformations. In *Proceedings of the Seventh Workshop on Languages and Compilers for Parallel Computing*, pages 92–106. Springer-Verlag, August 1994.

[4] A. W. Lim and M. S. Lam. Maximizing parallelism and minimizing synchronization with affine transforms. In *Conference Record of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Paris, France, January 1997.

[5] A. W. Lim and M. S. Lam. Maximizing parallelism and minimizing synchronization with affine partitions. *Parallel Computing*, 24(3–4):445–475, May 1998.

[6] A. W. Lim, S.-W. Liao, and M. S. Lam. Blocking and array contraction across arbitrarily nested loops using affine partitioning. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 102–112, June 2001.

[7] A. Schrijver. *Theory of Linear and Integer Programming*. Wiley, Chichester, 1986.

[8] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 30–44, June 1991.

[9] M. E. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *Transactions on Parallel and Distributed Systems*, 2(4):452–470, October 1991.

A Data Locality Optimizing Algorithm

Michael E. Wolf and Monica S. Lam

Computer Systems Laboratory
Stanford University, CA 94305

Abstract

This paper proposes an algorithm that improves the locality of a loop nest by transforming the code via interchange, reversal, skewing and tiling. The loop transformation algorithm is based on two concepts: a mathematical formulation of reuse and locality, and a loop transformation theory that unifies the various transforms as unimodular matrix transformations.

The algorithm has been implemented in the SUIF (Stanford University Intermediate Format) compiler, and is successful in optimizing codes such as matrix multiplication, successive over-relaxation (SOR), LU decomposition without pivoting, and Givens QR factorization. Performance evaluation indicates that locality optimization is especially crucial for scaling up the performance of parallel code.

1 Introduction

As processor speed continues to increase faster than memory speed, optimizations to use the memory hierarchy efficiently become ever more important. Blocking [9] or tiling [18] is a well-known technique that improves the data locality of numerical algorithms [1, 6, 7, 12, 13]. Tiling can be used for different levels of memory hierarchy such as physical memory, caches and registers; multi-level tiling can be used to achieve locality in multiple levels of the memory hierarchy simultaneously.

To illustrate the importance of tiling, consider the example of matrix multiplication:

```
for $I_1$ := 1 to $n$
  for $I_2$ := 1 to $n$
    for $I_3$ := 1 to $n$
      C[$I_1$,$I_3$] += A[$I_1$,$I_2$] * B[$I_2$,$I_3$];
```

In this code, although the same row of C and B are reused in the next iteration of the middle and outer loop, respectively, the large volume of data used in the intervening iterations may replace the data from the register file or the cache before it can be reused. Tiling reorders the execution sequence such that iterations from loops of the outer dimensions are executed before completing all the iterations of the inner loop. The tiled matrix multiplication is

```
for $II_2$ := 1 to $n$ by $s$
  for $II_3$ := 1 to $n$ by $s$
    for $I_1$ := 1 to $n$
      for $I_2$ := $II_2$ to min($II_2 + s - 1$,$n$)
        for $I_3$ := $II_3$ to min($II_3 + s - 1$, $n$)
          C[$I_1$,$I_3$] += A[$I_1$,$I_2$] * B[$I_2$,$I_3$];
```

Tiling reduces the number of intervening iterations and thus data fetched between data reuses. This allows reused data to still be in the cache or register file, and hence reduces memory accesses. The tile size s can be chosen to allow the maximum reuse for a specific level of memory hierarchy.

The improvement obtained from tiling can be far greater than from traditional compiler optimizations. Figure 1 shows the performance of 500 × 500 matrix multiplication on an SGI 4D/380 machine. The SGI 4D/380 has eight MIPS/R3000 processors running at 33 Mhz. Each processor has a 64 KB direct-mapped first-level cache and a 256 KB direct-mapped second-level cache. We ran four different experiments: without tiling, tiling to reuse data in caches, tiling to reuse data in registers [5], and tiling for both register and caches. For cache tiling, the data are copied into consecutive locations to avoid cache interference [12].

Tiling improves the performance on a single processor by a factor of 2.75. The effect of tiling on multiple processors is even more significant since it not only reduces the average data access latency but also the required memory bandwidth. Without cache tiling, contention over the

This research was supported in part by DARPA contract N00014-87-K-0828.

Proceedings of the ACM SIGPLAN '91 Conference on
Programming Language Design and Implementation.
Toronto, Ontario, Canada, June 26-28, 1991

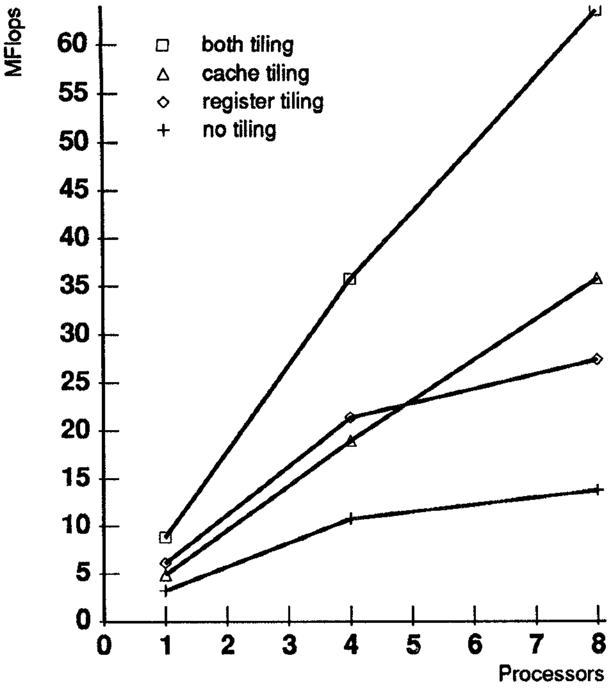


Figure 1: Performance of 500 × 500 double precision matrix multiplication on the SGI 4D/380. Cache tiles are 64 × 64 iterations and register tiles are 4 × 2.

memory bus limits the speedup to about 4.5 times. Cache tiling permits speedups of over seven for eight processors, achieving an impressive speed of 64 MFLOPS when combined with register tiling.

1.1 The Problem

The problem addressed in this paper is the use of loop transformations such as interchange, skewing and reversal to improve the locality of a loop nest. Matrix multiplication is a particularly simple example because it is both *legal* and *advantageous* to tile the entire nest. In general, it is not always possible to tile the entire loop nest. Some loop nests may not be tilable. Sometimes it is necessary to apply transformations such as interchange, skewing and reversal to produce a set of loops that are both tilable and advantageous to tile.

For example, consider the example of an abstraction of hyperbolic PDE code in Figure 2(a). Suppose the array in this example is larger than the memory hierarchy level of interest; the entire array must be fetched anew for each iteration of the outermost loop. Due to dependences, the loops must first be skewed before they can be tiled. This is also equivalent to finding non-rectangular tiles. Figure 2 contains the entire derivation of the tiled code, which we will use to illustrate our locality algorithm in the rest of the paper.

There are two major representations used in loop transformations: distance vectors and direction vectors [2, 17]. Loops whose dependences can be summarized by distance vectors are special in that it is advantageous, possible and easy to tile all loops [10, 15]. General loop nests, whose dependences are represented by direction vectors, may not be tilable in their entirety. The data locality problem addressed in this paper is to find the best combination of loop interchanges, skewing, reversal and tiling that maximizes the data locality within loop nests, subject to the constraints of direction and distance vectors.

Research has been performed on both the legality and the desirability of loop transformations with respect to data locality. Early research on optimizing loops with direction vectors concentrated on the legality of pairwise transformations, such as when it is legal to interchange a pair of loops. However, in general, it is necessary to apply a series of primitive transformations to achieve goals such as parallelism and data locality. This has led to work on combinations of primitive transforms. For example, Wolfe [18] shows how to determine when a loop nest can be tiled; two-dimensional tiling can be achieved via a pair of transformations known as "strip-mine and interchange" [14] or "unroll and jam" [5]. Wolfe also shows that skewing can make a pair of loops tilable. Banerjee discusses general unimodular transforms for two-deep loop nests [4]. A technique used in practice to handle general n-dimensional loop nests is to determine *a priori* the sequence of loop transforms to attempt. This technique is inadequate because certain transformations, such as loop skewing, may not improve code, but may enable other optimizations that do so. Which of these to perform will depend on which other optimizations will be enabled: the desirablity of a transformation cannot be evaluated locally. Furthermore, the correct ordering of optimizations are highly program dependent.

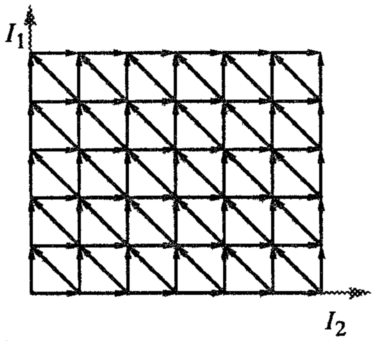
On the desirability of tiling, previous work concentrated on how to determine the cache performance and tune the loop parameters *for a given loop nest.* Porterfield gives an algorithm for estimating the hit rate of a fully-associative LRU (least recently used replacement policy) cache of a given size [14]. Gannon et al. uses *reference windows* to determine the minimum memory locations necessary to maximize reuse in a loop nest [8]. These evaluation functions are useful for comparing the locality performance after applying transformations, but do not suggest the transformations to apply when a series of transformations may first need to be applied before tiling becomes feasible and useful.

If we were to use these evaluation functions to find the suitable transformations, we would need to search the transformation space exhaustively. The previously proposed method of enumerating the all possible combinations of legal transformations is expensive and not even possible if there are infinitely many combinations, as is the case when we include skewing.

(a): Extract dependence information

```
for $I_1$ := 0 to 5 do
  for $I_2$ := 0 to 6 do
    A[$I_2+1$] := 1/3 * (A[$I_2$] + A[$I_2+1$] + A[$I_2+2$]);
```

$$D = \{(0,1),(1,0),(1,-1)\}.$$



(b): Extract locality information

Uniformly generated set = {A[I_2],A[I_2+1],A[I_2+2]}.

reuse category	reuse vector space	potential reuse factor
self-temporal	span$\{(1,0)\}$	s
self-spatial	span$\{(1,0),(0,1)\}$	l
group	span$\{(1,0),(0,1)\}$	3

Loops carrying reuse = $\{I_1, I_2\}$.

(c): Search transformation space

localized space	transformation	sources of locality	accesses per iteration
span$\{(0,1)\}$	$T = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$	self-spatial, group	$1/l$
span$\{(1,0)\}$	not possible	—	—
span$\{(1,0),(0,1)\}$	$T = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$	self-temporal, self-spatial, group	$1/(ls)$

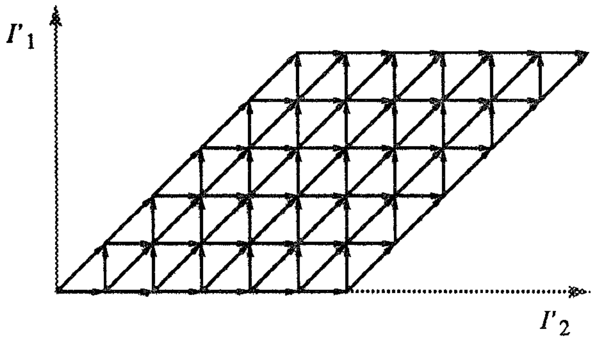
The best legal choice is to tile both I_1 and I_2.

(d): Skew to make inner loop nest fully permutable

```
for $I_1'$ := 0 to 5 do
  for $I_2'$ := $I_1'$ to 6+$I_1'$ do
    A[$I_2' - I_1' + 1$] := 1/3 * (A[$I_2' - I_1'$]
        + A[$I_2' - I_1' + 1$] + A[$I_2' - I_1' + 2$]);
```

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

$$D' = TD = \{(0,1),(1,1),(1,0)\}$$



(e): Final code

```
for $II_2'$ := 0 to 11 by 2 do
  for $I_1'$ := 0 to 5 do
    for $I_2'$ := max($I_1'$, $II_2'$) to min(6+$I_1'$, $II_2'+1$) do
      A[$I_2' - I_1' + 1$] := 1/3 * (A[$I_2' - I_1'$]
          + A[$I_2' - I_1' + 1$] + A[$I_2' - I_1' + 2$]);
```

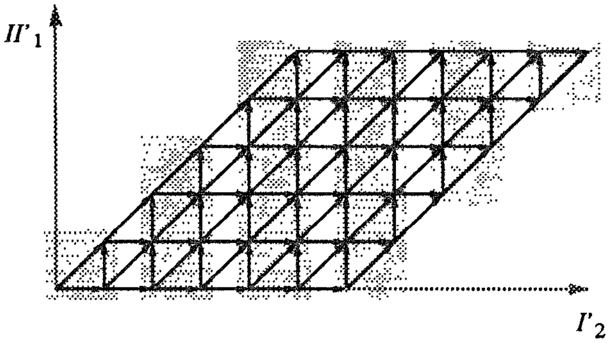


Figure 2: Example of the locality optimization algorithm on a hyperbolic PDE-style loop nest.

1.2 An Overview

This paper focuses on maximizing data locality at the cache level. Although the basic principles in memory hierarchy optimization are similar for all levels, each level has slightly different characteristics, requiring slightly different considerations. Caches usually have small set associativity, so cache data conflicts can cause desired data to be replaced. We have found that the performance of tiled code fluctuates dramatically with the size of the data matrix, due to cache interference [12]. We show that this effect can be mitigated by copying reused data to consecutive locations before the computation, or choosing the tile size according to the matrix size. Both of these optimizations can be performed after code transformation, and thus cache interference need not be considered at code transformation time.

Another major difference between caches and registers is their capacity. To fit all the data used in a tile into the faster level of memory hierarchy, transformations that increase the dimensionality of the tile may reduce the tile size. However, as we will show, the reduction of memory accesses can be a factor of s^d when d-dimensional tiles of lengths s are used. Thus for typical cache sizes and loop depths found in practice, increasing the dimensionality of the tile will reduce the total number of memory access, even though the length of a tile side may be smaller for larger dimensional tiles. Thus the choice of tile size can be postponed until after the application of optimizations to increase the dimensionality of locality.

The problem addressed in this paper is the choice of loop transforms to increase data locality. We describe a locality optimization algorithm that applies a combination of loop interchanges, skewing, reversal and tiling to improve the data locality of loop nests. The analysis is applicable to array references whose indices are affine functions of the loop indices. The transformations are applicable to loops with not just distance dependence vectors, but also direction vectors as well. Our locality optimization is based on two results: a new transformation theory and a mathematical formulation of data locality.

Our loop transformation theory unifies common loop transforms including interchange, skewing, reversal and their combinations, as unimodular matrix transforms. This unification reduces the legality of all compound transformations to satisfying the same simple constraints. In this way, a loop transformation problem can be formulated as solving for the transformation matrix that maximizes an objective function, subjected to a set of constraints. This matrix model has previously been applied only to distance vectors. We have extended the framework to handle direction vectors as well.

The second result is a formulation of an objective function for data locality. We introduce the concepts of a *reuse vector space* to capture the potential of data locality optimization for a given loop nest. This formulation collapses the space of transformed code into equivalence classes, hence allowing pruning of the search for the best transformation.

Our locality algorithm uses the evaluation function and the legality constraints to reduce the search space of the transforms. Unfortunately, finding the optimal transformation still requires a complex algorithm that is exponential in the loop nest depth. We have devised a heuristic algorithm that works well for common cases found in practice.

We have implemented the algorithm in the SUIF (Stanford University Intermediate Format) compiler. Our algorithm applies unimodular and tiling transforms to loop nests, handles non-rectangular loop bounds, and generates non-uniform tiles to handle non-perfectly loop nests. It is successful in tiling numerical algorithms such as matrix multiplication, successive over-relaxation (SOR), LU decomposition without pivoting, and Givens QR factorization. For simplicity, we assume here that all loops are perfectly nested. That is, all computation is nested in the innermost loop.

In Section 2, we discuss our dependence representation and the basics of unimodular transformations. How tiling takes advantage of reuse in an algorithm is discussed in Section 3. In Section 4, we describe how to identify and evaluate reuse in a loop nest. We use those results to formulate an algorithm to improve locality. Finally, we present some experimental data on tiling for a cache.

2 A Loop Transformation Theory

While individual loop transformations are well understood, ad hoc techniques have typically been used in combining them to achieve a particular goal. Our loop transformation theory offers a foundation for deriving compound transformations efficiently [16]. We have previously shown the use of this theory to maximizing the degree of parallelism in a loop nest; we will demonstrate its applicability to data locality in this paper.

2.1 The Iteration Space

In this model, a loop nest of depth n corresponds to a finite convex polyhedron of iteration space $\mathcal{Z}^n$, bounded by the loop bounds. Each iteration in the loop corresponds to a *node* in the polyhedron, and is identified by its index vector $\vec{p} = (p_1, p_2, \ldots, p_n)$; p_i is the loop index of the i loop in the nest, counting from the outermost to innermost loop. The iterations are therefore executed in lexicographic order of their index vectors. That is, if $\vec{p_2}$ is lexicographically greater than $\vec{p_1}$, written $\vec{p_2} \succ \vec{p_1}$, iteration $\vec{p_2}$ executes after iteration $\vec{p_1}$.

Our dependence representation is a generalization of distance and direction vectors. A dependence vector in an n-nested loop is denoted by a vector $\vec{d} = (d_1, d_2, \ldots, d_n)$. Each component d_i is a possibly infinite range of integers, represented by $[d_i^{\min}, d_i^{\max}]$, where

$$d_i^{\min} \in \mathcal{Z} \cup \{-\infty\},\ d_i^{\max} \in \mathcal{Z} \cup \{\infty\} \text{ and } d_i^{\min} \le d_i^{\max}.$$

A single dependence vector therefore represents a set of distance vectors, called its *distance vector set*:

$$\mathcal{E}(\vec{d}) = \left\{(e_1,\ldots,e_n)|e_i \in \mathcal{Z} \text{ and } d_i^{\min} \le e_i \le d_i^{\max}\right\}.$$

Each of the distance vector defines a set of edges on pairs of nodes in the iteration space. Iteration $\vec{p_2}$ depends on iteration $\vec{p_1}$, and thus must execute after $\vec{p_1}$, if for some distance vector $\vec{e}$, $\vec{p_2} = \vec{p_1} + \vec{e}$. By definition, since $\vec{p_2} \succ \vec{p_1}$, $\vec{e}$ must therefore be lexicographically greater than $\vec{0}$, or simply, lexicographically positive.

The dependence vector $\vec{d}$ is also a distance vector if each of its components is a degenerate range consisting of a singleton value, that is, $d_i^{\min} = d_i^{\max}$. For short, we simply denote such a range with the value itself. There are three common ranges found in practice: $[1,\infty]$ denoted by '+', $[-\infty,-1]$ denoted by '−', and $[-\infty,\infty]$ denoted by '±'. They correspond to the previously defined directions of '<', '>', and '∗', respectively [17].

We have extended the definition of vector operations to allow for ranges in each of the component. In this way, we can manipulate a combination of distances and directions simply as vectors. This is needed to support the matrix transform model, discussed below.

Our model differs from the previously used model in that all our dependence vectors are represented as lexicographically positive vectors. In particular, consider a strictly sequential pair of loops such as the one below:

```
for $I_1$ := 0 to $n$ do
  for $I_2$ := 0 to $n$ do
    $b$ := $g(b)$;
```

The dependence of this program would previously be represented as ('∗','∗'). In our model, we represent them as a pair of lexicographically positive vectors, (0,'+'),('+','±'). The requirement that all dependences are lexicographically greatly simplifies the legality tests for loop transformations. The dependence vectors define a partial order on the nodes in the iteration space, and any topological ordering on the graph is a legal execution order, as all dependences in the loop are satisfied.

2.2 Unimodular Loop Transformations

With dependences represented as vectors in the iteration space, loop transformations such as interchange, skewing and reversal, can be represented as matrix transformations.

Let us illustrate the concept with the simple example of an interchange on a loop with distances. A loop interchange transformation maps iteration (p_1,p_2) to iteration $(p_{2,1})$. In matrix notation, we can write this as

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} p_1 \\ p_2 \end{bmatrix} = \begin{bmatrix} p_2 \\ p_1 \end{bmatrix}.$$

The elementary permutation matrix $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ thus performs the loop interchange transformation on the iteration space.

Since a matrix transformation T is a linear transformation on the iteration space, $T\vec{p_2} - T\vec{p_1} = T(\vec{p_2} - \vec{p_1})$. Therefore, if $\vec{d}$ is a distance vector in the original iteration space, then $T\vec{d}$ is a distance vector in the transformed iteration space. Thus in loop interchange, the dependence vector (d_1,d_2) is mapped into

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} d_1 \\ d_2 \end{bmatrix} = \begin{bmatrix} d_2 \\ d_1 \end{bmatrix}.$$

in the transformed space. Therefore if the transformed dependence vector remains lexicographically positive, the interchange is legal.

The loop reversal and skewing transform can similarly be represented as matrices [3, 4, 16]. (An example of skewing is shown in Figure 2(d).) These matrices are unimodular matrices, that is, they are square matrices with integral components and a determinant of one or negative one. Because of these properties, the product of two unimodular matrices is unimodular, and the inverse of a unimodular matrix is unimodular, so that combinations of unimodular loop transformations and inverses of unimodular loop transformations are also unimodular loop transformations. Under this formulation, there is a simple legality test for all transforms.

Theorem 2.1 . *Let D be the set of distance vectors of a loop nest. A unimodular transformation T is legal if and only if $\forall \vec{d} \in D : T\vec{d} \succ \vec{0}$.*

The elegance of this theory helps reduce the complexity of the implementation. Once the dependences are extracted, the derivation of the compound transform simply consists of matrix and vector operations. After the transformation is determined, a straightforward algorithm applies the transformation to the loop bounds and derives the final code.

3 The Localized Vector Space

It is important to distinguish between *reuse* and *locality*. We say that a data item is *reused* if the same data is used in multiple iterations in a loop nest. Thus reuse is a measure that is inherent in the computation and not dependent on the particular way the loops are written. This reuse may not lead to saving a memory access if intervening iterations flush the data out of the cache between uses of the data.

For example, reference `A[`I_2`]` in Figure 3 touches different data within the innermost loop, but reuses the same elements across the outer loop. More precisely, the same data `A[`I_2`]` is used in iterations $(I_1,I_2), 1 \le I_1 \le n$. There is reuse, but the reuse is separated by accesses to $n-1$

```
for $I_1$ := 1 to $n$ do
  for $I_2$ := 1 to $n$ do
    $f$(A[$I_1$],A[$I_2$]);
```

Figure 3: A simple example.

other data. When n is large, the data is removed from the cache before it can be reused, and there is no locality. Therefore, *a reuse does not guarantee locality.*

Specifically, if the innermost loop contains a large number of iterations and touches a large number of data, only the reuse within the innermost loop can be exploited. We can apply a unimodular transformation to improve the amount of data reused in the innermost loop. However, as shown in this example, reuse can sometime occur along multiple dimensions of the iteration space. To exploit multi-dimensional reuse, unimodular transformations must be coupled with tiling.

3.1 Tiling

In general, tiling transforms an n-deep loop nest into a $2n$-deep loop nest where the inner n loops execute a compiler-determined number of iterations. Figure 4 shows the code after tiling the example in Figure 2(a), using a tile size of 2×2. The two innermost loops execute the iterations within each tile, represented as 2×2 squares in the figure. The two outer loops, represented by the two axes in the figure, execute the 12 tiles. As the outer loop nests of tiled code controls the execution of the tiles, we will refer to them as the *controlling loops.* When we say tiling, we refer to the partitioning of the iteration space into rectangular blocks. Non-rectangular blocks are obtained by first applying unimodular transformations to the iteration space and then applying tiling.

Like all transformations, it is not always possible to tile. Loops I_i through I_j in a loop nest can be tiled if they are *fully permutable* [11, 16]. Loops I_i through I_j in a loop nest are fully permutable if and only if all dependence vectors are lexicographically positive and for each dependence vector, either $(d_1, \ldots, d_{i-1})$ is lexicographically positive, or the ith through jth components of $\vec{d}$ are all non-negative. For example, the components of dependences in Figure 2(b) are all non-negative, and the two loops are therefore fully permutable and tilable. Full permutability is also very useful for improving parallelism [16], so parallelism and locality are often compatible goals.

After tiling, both groups of loops, the loops within a tile and the loops controlling the tiles, remain fully permutable. For example, in Figure 4, loops II'_1 and II'_2 can be interchanged, and so can I'_1 and I'_2. By interchanging I'_1 and I'_2, the loops II'_2 and I'_2 can be trivially coalesced to produce the code in Figure 2(e). This transformation has previously been known as "strip-mine and interchange". "Unroll and jam" is yet another equivalent form to tiling. Since all loops within a tile are fully permutable, any loop in an n-dimensional tile can be chosen to be the coalesced loop.

3.2 Tiling for Locality

When we tile two innermost loops, we execute only a finite number of iterations in the innermost loop before executing iterations from the next loop. The tiled code for the example in Figure 3 is:

```
for $II_2$ := 1 to $n$ by $s$ do
  for $I_1$ := 1 to $n$ do
    for $I_2$ := $II_2$ to max($n$, $II_2 + s - 1$) do
      $f$(A[$I_1$],A[$I_2$]);
```

We choose the tile size such that the data used within the tile can be held within the cache. In this example, as long as s is smaller than the cache size (in words), A[I_2] will still be present in the cache when it is reused. Thus tiling increases the number of dimensions in which reuse can be exploited. We call the iterations that can exploit reuse the *localized iteration space.* In the example above, reuse is exploited for loops I_1 and I_2 only, so the localized iteration space includes only those loops.

The depth of loop nesting and the number of variables accessed within a loop body are small compared to typical cache sizes. Therefore we should always be able to choose suitable tile sizes such that the reused data can be stored in a cache. Since we do not need the tile size to determine if reuse is possible, we abstract away the loop bounds of the localized iteration space, and characterize the localized iteration space as a *localized vector space.* Thus we say that tiling the loop nest of Figure 3 results in a localized vector space of $\text{span}\{(0,1),(1,0)\}$.

In general, if n is the first loop with a large bound, counting from innermost to outermost, then reuse occurring within the inner n loops can be exploited. Therefore the localized vector space of a tiled loop is simply that of the innermost tile, whether the boundary between the controlling loops and the loops within the tile be coalesced or not.

4 Evaluating Reuse and Locality

Since unimodular transformations and tiling can modify the localized vector space, knowing where there is reuse in the iteration space can help guide the search for the transformation that delivers the best locality. Also, to choose between alternate transformations that exploit different reuses in a loop nest, we need a metric to quantify locality for a specific localized iteration space.

```
for $II_1'$ := 0 to 5 by 2 do
  for $II_2'$ := 0 to 11 by 2 do
    for $I_1'$ := $II_1'$ to min($I_1'+1$, 5) do
      for $I_2'$ := max($II_1'$, $II_2'$) to min(6+$I_1'$, $II_2'$+1) do
        A[$I_2'$] := 1/3 * (A[$I_2'-1$] + A[$I_2'$] + A[$I_2'+1$]);
```

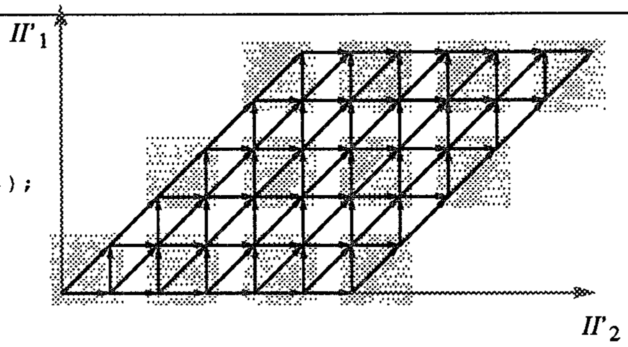


Figure 4: Iteration space and dependences of tiled code from Figure 2(a).

4.1 Types of Reuse

Reuse occurs when a reference within a loop accesses the same data location in different iterations. We call this *self-temporal reuse*. Likewise, if a reference accesses data on the same cache line in different iterations, it is said to possess *self-spatial reuse*. Furthermore, different references may access the same locations. We say that there is *group-temporal reuse* if the references refer to the same location, and *group-spatial reuse* if they refer to the same cache line. Examples of each type of reuse are given below.

Let us first consider reuse within a single reference. In Figure 3, the reference A[I_1] has self-temporal reuse in the innermost loop because it accesses the same element for all iterations (I_1, I_2), where $1 \leq I_2 \leq n$. Similarly, A[I_2] has temporal reuse in the outermost loop. In both cases, the same data is reused n times; that is, the memory accesses are reduced to $1/n$th of the original if the reuse is exploited.

Besides self-temporal reuse, A[I_2] also has self-spatial reuse in the innermost loop, since each cache line is reused l times, where l is the cache line size. Likewise, A[I_1] has self-spatial locality in the outermost loop. Altogether, each reference has temporal reuse of a factor of n, and an additional spatial reuse of a factor of l. Therefore, in either case each cache line can be reused nl times if the localized space encompasses all the reuse.

By definition, temporal reuse is a subset of spatial reuse; reusing the same location is trivially reusing the same cache line. That is, loops carrying temporal reuse also carry spatial reuse. While the same data can be reused arbitrarily many times depending on the program, the additional factor of improvement from spatial reuse is limited to a factor of l, where l is the cache line size.

We now consider group reuse, reuse among different references. Trivially, identical references within the same loop nest will result in the same cache behavior as if there had just been one reference. Now consider, for example, the references A[I_2], A[I_2+1] and A[I_2+2] in Figure 2. In addition to any self reuse these references might have, it is easy to see that they have a factor of three reuse in the I_2 loop.

In contrast, consider the example in Figure 3. In this 2-dimensional iteration space, the iterations that use the same data between the two groups are the kth column and the kth row. As it is, only iterations near the diagonal of the space can exploit locality. Furthermore, no unimodular or tiling transformation can place uses of the same data close to each other. Thus, multiple references to the same array do not necessarily result in significant locality.

The difference between the A[I_2], A[I_2+1], A[I_2+2] references and the A[I_1], A[I_2] references is that the former set of references has similar array index functions, differing only in the constant term. Such references are known as *uniformly generated references*. The concept of uniformly generated references is also used by Gannon et al. [8] in estimating their reference windows.

Definition 4.1 *Let n be the depth of a loop nest, and d be the dimensions of an array* A. *Two references* A$[\vec{f}(\vec{\imath})]$ *and* A$[\vec{g}(\vec{\imath})]$, *where $\vec{f}$ and $\vec{g}$ are indexing functions $Z^n \rightarrow Z^d$, are called* uniformly generated *if*

$$\vec{f}(\vec{\imath}) = H\vec{\imath} + \vec{c}_f \text{ and } \vec{g}(\vec{\imath}) = H\vec{\imath} + \vec{c}_g$$

where H is a linear transformation and $\vec{c}_f$ and $\vec{c}_g$ are constant vectors.

Since little exploitable reuse exists between non-uniformly generated references, we partition references in a loop nest into equivalence classes of references that operate on the same array and have the same H. We call these equivalence classes *uniformly generated sets*. In the degenerate case where a uniformly generated set consists of only one element, we have only self reuse for that reference. In the example in Figure 2, the indexing functions of the references A[I_2], A[I_2+1] and A[I_2+2] can be written as

$$\begin{bmatrix} 0 & 1 \end{bmatrix} \begin{bmatrix} I_1 \\ I_2 \end{bmatrix} + \begin{bmatrix} 0 \end{bmatrix},$$

$$\begin{bmatrix} 0 & 1 \end{bmatrix} \begin{bmatrix} I_1 \\ I_2 \end{bmatrix} + \begin{bmatrix} 1 \end{bmatrix},$$

$$\begin{bmatrix} 0 & 1 \end{bmatrix} \begin{bmatrix} I_1 \\ I_2 \end{bmatrix} + \begin{bmatrix} 2 \end{bmatrix}$$

respectively. These references belong to a single uniformly generated set with an H of $\begin{bmatrix} 0 & 1 \end{bmatrix}$.

4.2 Quantifying Reuse and Locality

So far, we have discussed intuitively where reuse takes place. We now show how to identify and quantify reuse within an iteration space. We use vector spaces to represent the directions in which reuse is found; these are the directions we wish to include in the localized space. We evaluate the locality available within a loop nest using the metric of memory accesses per iteration of the innermost loop. A reference with no locality will result in one access per iteration.

4.2.1 Self-Temporal

Consider the self-temporal reuse for a reference A[$H\vec{\imath}+\vec{c}$]. Iterations $\vec{\imath}_1$ and $\vec{\imath}_2$ reference the same data element whenever $H\vec{\imath}_1+\vec{c} = H\vec{\imath}_2+\vec{c}$, that is, when $H(\vec{\imath}_1-\vec{\imath}_2) = \vec{0}$. We say that there is reuse in *direction* $\vec{r}$ when $H\vec{r} = \vec{0}$. That is, reuse is exploited if $\vec{r}$ is included in the localized vector space. The solution to this equation is $\ker H$, a vector space in $\mathcal{R}^n$. We call this the *self-temporal reuse vector space*, R_{ST}. Thus,

$$R_{ST} = \ker H.$$

For example, the reference C[I_1, I_3] in the matrix multiplication in Section 1 produces:

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

$$R_{ST} = \ker H = \text{span}\{(0,1,0)\}.$$

Informally, we say there is self-temporal reuse in loop I_2. Since loop I_2 has n iterations, there are n reuses of C in this loop nest. Similar analysis shows that A[I_1, I_2] has self-temporal reuse in the I_3 direction and B[I_2, I_3] has reuse in the I_1 direction.

In this example nest, every reuse vector space is one-dimensional. In general, the reuse vector space can have zero or more dimensions. If the dimensionality is zero, then $R_{ST} = \emptyset$ and there is no self-temporal reuse. An example of a two-dimensional reuse vector space is the reference $A[I_1]$ within a three-deep nest with loops (I_1, I_2, I_3). In general, if the number of iterations executed along each dimension of reuse is s, then each element is reused $s^{\dim(R_{ST})}$ times.

As discussed in Section 3, a reuse is exploited only if it occurs within the localized vector space. Thus, a reference has self-temporal locality if its self-temporal reuse space R_{ST} and the localized space L in the code have a non-null intersection. The dimensionality of $R_{ST} \cap L$ indicates the quantity of self-temporal reuse utilized: the number of memory accesses is, simply,

$$1/s^{\dim(R_{ST}\cap L)},$$

where s is the number of iterations in each dimension.

Consider again the matrix multiplication code. The only localized direction in the untiled code is the innermost loop, which is span$\{(0,0,1)\}$. It coincides exactly with R_{ST}(A[I_1, I_2]), resulting in locality for that reference. Similarly, the empty intersection with the reuse vector space of references B[I_2, I_3] and C[I_1, I_3] indicates no temporal locality for these references. In contrast, the localized vector space of the tiled matrix multiplication spans all the three loop directions. Trivially, there is a non-empty intersection with each of the reuse spaces, so self-temporal reuses are exploited for all references.

All references within a single uniformly generated set have the same H, and thus the same self-temporal reuse vector space. Therefore the derivation of the reuse vector spaces and their intersections with the localized space need only be performed once per uniformly generated set, not once per reference. The total number of memory accesses is the sum over that of each reference. For typical values of tile sizes chosen for caches, the number of memory accesses will be dominated by the one with the smallest values of $\dim(R_{ST}\cap L)$. Thus the goal of loop transforms is to maximize the dimensionality of reuse; the exact value of the tile size s does not affect the choice of the transformation.

4.2.2 Self-Spatial

Without loss of generality, we assume that data is stored in row major order. Spatial reuse can occur only if accesses are made to the same row. Furthermore, the difference in the row index expression has to be within the cache line size. For a reference such as A[I_1, I_2], the memory accesses in the I_2 loop will be reduced by a factor of l, where l is the line size. However, there is no reuse for A[I_1, 10*I_2] if the cache line is less than 10 words. For any stride $k : 1 \leq k \leq l$, the potential reuse factor is l/k.

All but the row index must be identical for a reference A[$H\vec{\imath}+\vec{c}$] to possess self-spatial reuse. We let H_S be H with all elements of the last row replaced by 0. The *self-spatial reuse vector space* is simply

$$R_{SS} = \ker H_S.$$

As discussed above, temporal locality is a special case of spatial locality. This is reflected in our mathematical formulation since

$$\ker H \subset \ker H_S.$$

If $R_{SS}\cap L = R_{ST}\cap L$, the reuse occurs on the same word, the spatial reuse is also temporal, and there is no additional gain due to the prefetching of cache lines. If $R_{SS} \cap L \neq R_{ST} \cap L$, however, different elements in the same row are reused. If the transformed array reference is of stride one, all the data in the cache line is also reused, resulting in a total of $1/(l s^{\dim(R_{ST} \cap L)})$ memory accesses per iteration. If the stride is $k < l$, the number of accesses per iteration is $k/(l s^{\dim(R_{ST} \cap L)})$. As an example, the locality for the original and fully tiled matrix multiplication is tabulated in Table 1.

4.2.3 Group-Temporal

To illustrate the analysis on group-temporal reuse, we use the code for a single SOR relaxation step:

```
for $I_1$ := 1 to $n$ do
  for $I_2$ := 1 to $n$ do
    A[$I_1$,$I_2$] := 0.2*(A[$I_1$,$I_2$]+A[$I_1+1$,$I_2$]
            +A[$I_1-1$,$I_2$]+A[$I_1$,$I_2+1$]
            +A[$I_1$,$I_2-1$]);
```

There are five distinct references in this innermost loop, all to the same array. The reuse between these references, can potentially reduce the number of accesses by a factor of 5. If all temporal and spatial reuses are exploited, the total number of memory accesses per iteration is reduced from 5 to $1/l$, where l is the cache line size. With the way the nest is currently written, the localized space consists of only the I_2 direction. The reference A[I_1,I_2-1] uses the same data as A[I_1,I_2] from the previous iteration, and A[I_1,I_2+1] from the second previous iteration. However, A[I_1-1,I_2] and A[I_1+1,I_2] must necessarily each access a different set of data. The number of accesses per iteration is thus $3/l$. We now show how to mathematically calculate and factor in this group-temporal reuse.

As discussed above, group-temporal reuse need only be calculated for elements within the same uniformly generated set. Two distinct references A[$H\vec{\imath}+\vec{c}_1$] and A[$H\vec{\imath}+\vec{c}_2$] have *group temporal* reuse within a localized space L if and only if

$$\exists \vec{r} \in L : H\vec{r} = \vec{c}_1 - \vec{c}_2.$$

To determine whether such an $\vec{r}$ exists, we solve the system of equations $H\vec{r} = \vec{c}_1 - \vec{c}_2$ to get a particular solution $\vec{r}_p$, if one exists. The general solution is $\ker H + \vec{r}_p$, and so there exists a $\vec{r}$ satisfying the above equation if and only if

$$(\operatorname{span}\{\vec{r}_p\} + \ker H) \cap L \neq \ker H \cap L.$$

In the SOR example above, H is the identity matrix and $\ker H = \emptyset$. Thus there is group reuse between two references in L if and only if $\vec{r_p} = \vec{c}_1 - \vec{c}_2 \in L$. When the inner loop is I_2, there is reuse between A[I_1,I_2-1] and A[I_1,I_2], since

$$(0,-1) - (0,0) \in \operatorname{span}\{(0,1)\}.$$

Reuse does not exist between A[I_1,I_2-1] and A[I_1+1,I_2], since

$$(0,-1) - (1,0) \notin \operatorname{span}\{(0,1)\}.$$

In fact, the references fall into three equivalence classes:

$$\{\texttt{A}[I_1, I_2-1], \texttt{A}[I_1, I_2], \texttt{A}[I_1, I_2+1]\}$$
$$\{\texttt{A}[I_1+1, I_2]\}$$
$$\{\texttt{A}[I_1, I_2]\}$$

Reuse exists and only exists between all pairs of references within each class. Thus, the effective number of memory references is simply the number of equivalence classes. In this case, there are three references instead of five, as expected.

In general, the vector space in which there is any group-temporal reuse R_{GT} for a uniformly generated set with references $\{\texttt{A}[H\vec{\imath}+\vec{c_1}], \ldots, \texttt{A}[H\vec{\imath}+\vec{c_g}]\}$ is defined to be

$$R_{GT} = \operatorname{span}\{\vec{r}_2, \ldots, \vec{r}_g\} + \ker H$$

where for $k = 2, \ldots, g$, $\vec{r}_k$ is a particular solution of

$$H\vec{r} = \vec{c}_1 - \vec{c}_k.$$

For a particular localized space L, we get an additional benefit due to group reuse if and only if $R_{GT} \cap L \neq R_{ST} \cap L$. To determine the benefit, we must partition the references into equivalence classes as shown above. Denoting the number of equivalence classes by g_T, the number of memory references for the uniformly generated set per iteration is g_T, instead of g.

4.2.4 Group-Spatial

Finally, there may also be *group-spatial* reuse. For example, in the following loop nest

```
for $I_1$ := 1 to $n$ do
  $f$(A[$I_1$,0],A[$I_1$,1]);
```

the two references refer to the same cache line in each iteration.

Following a similar analysis as above, the group-spatial vector space R_{GS} for a uniformly generated set with references $\{\texttt{A}[H\vec{\imath}+\vec{c_1}], \ldots, \texttt{A}[H\vec{\imath}+\vec{c_g}]\}$ is defined to be

$$R_{GS} = \operatorname{span}\{\vec{r}_2, \ldots, \vec{r}_g\} + \ker H_S$$

where H_S is H with all elements of the last row replaced by 0 and for $k = 2, \ldots, n$, $\vec{r}_k$ is a particular solution of

$$H\vec{r} = \vec{c}_{S,1} - \vec{c}_{S,k}$$

where $\vec{c}_{S,i}$ denotes $\vec{c}_i$ with the last row set to 0. The relationships between the various reuse vector spaces are therefore

$$R_{ST} \subset R_{SS}, R_{GT} \subset R_{GS}$$

Reference	*Reuse*		*Untiled* ($L = span\{\vec{e}_3\}$)			*Tiled* ($L = span\{\vec{e}_1, \vec{e}_2, \vec{e}_3\}$)		
	R_{ST}	R_{SS}	$R_{ST} \cap L$	$R_{SS} \cap L$	*cost*	$R_{ST} \cap L$	$R_{SS} \cap L$	*cost*
A[I_1, I_2]	span$\{\vec{e}_3\}$	span$\{\vec{e}_2, \vec{e}_3\}$	span$\{\vec{e}_3\}$	span$\{\vec{e}_3\}$	$1/s$	span$\{\vec{e}_3\}$	span$\{\vec{e}_2, \vec{e}_3\}$	$1/(ls)$
B[I_2, I_3]	span$\{\vec{e}_1\}$	span$\{\vec{e}_1, \vec{e}_3\}$	$\emptyset$	span$\{\vec{e}_3\}$	$1/l$	span$\{\vec{e}_1\}$	span$\{\vec{e}_1, \vec{e}_3\}$	$1/(ls)$
C[I_1, I_3]	span$\{\vec{e}_2\}$	span$\{\vec{e}_2, \vec{e}_3\}$	$\emptyset$	span$\{\vec{e}_3\}$	$1/l$	span$\{\vec{e}_2\}$	span$\{\vec{e}_2, \vec{e}_3\}$	$1/(ls)$

Table 1: Self-temporal and self-spatial locality in tiled and untiled matrix multiplication. We use the vectors $\vec{e}_1$, $\vec{e}_2$ and $\vec{e}_3$ to represent $(1,0,0)$, $(0,1,0)$ and $(0,0,1)$ respectively.

Two references with index functions $H\vec{i}+\vec{c_i}$ and $H\vec{i}+\vec{c_j}$ belong to the same group-spatial equivalence class if and only if

$$\exists \vec{r} \in L : H_S\vec{r} = \vec{c}_{S,i} - \vec{c}_{S,j}.$$

We denote the number of equivalence sets by g_S. Thus, $g_S \leq g_T \leq g$, where g is the number of elements in the uniformly generated set. Using a probabilistic argument, the effective number of accesses per iteration for stride one accesses, with a line size l, is

$$g_S + (g_T - g_S)/l.$$

4.2.5 Combining Reuses

The union of the group-spatial reuse vector spaces of each uniformly generated set captures the entire space in which there is reuse within a loop nest. If we can find a transformation that can generate a localized space that encompasses these spaces, then all the reuses will be exploited. Unfortunately, due to dependence constraints, this may not be possible. In that case, the locality optimization problem is to find the transform that delivers the fewest memory accesses per iteration.

From the discussion above, the memory accesses per iteration for a particular transformed nest can be calculated as follows. The total number of memory accesses is the sum of the accesses for each uniformly generated set. The general formula for the number of accesses per iteration for a uniformly generated set, given a localized space L and line size l, is

$$\frac{g_S + (g_T - g_S)/l}{l^e s^{\dim(R_{SS} \cap L)}}$$

where

$$e = \begin{cases} 0 & R_{ST} \cap L = R_{SS} \cap L \\ 1 & \text{otherwise.} \end{cases}$$

We now study the full example in Figure 2(a) to illustrate the reuse and locality analysis. The reuse vector spaces of the code are summarized in Figure 2(b). For the uniformly generated set in the example,

$$H = \begin{bmatrix} 0 & 1 \end{bmatrix}, H_S = \begin{bmatrix} 0 & 0 \end{bmatrix}.$$

The reuse vector spaces are

$$\begin{aligned} R_{ST} &= \ker H = \text{span}\{(1,0)\} \\ R_{SS} &= \ker H_S = \text{span}\{(0,1),(1,0)\} \\ R_{GT} &= \text{span}\{(0,1)\} + \ker H = \text{span}\{(0,1),(1,0)\} \\ R_{GS} &= \text{span}\{(0,1)\} + \ker H_S = \text{span}\{(0,1),(1,0)\}. \end{aligned}$$

When the localized space L is span$\{(0,1)\}$,

$$\begin{aligned} R_{ST} \cap L &= \emptyset \\ R_{SS} \cap L &= \text{span}\{(0,1)\} \neq R_{ST} \\ R_{GT} \cap L &= \text{span}\{(0,1)\} \\ R_{GS} \cap L &= \text{span}\{(0,1)\} = R_{GT}. \end{aligned}$$

Since $g_T = 1$, the total number of memory accesses per iteration is $1/l$. Similar derivation shows the overall number of accesses per iteration for the localized space of span$\{(1,0),(0,1)\}$ to be $1/(ls)$.

The data locality optimization problem can now be formulated as follows:

Definition 4.2 *For a given iteration space with*

1. *a set of dependence vectors, and*
2. *uniformly generated reference sets*

the data locality optimization problem *is to find the unimodular and/or tiling transform, subject to data dependences, that minimizes the number of memory accesses per iteration.*

5 An Algorithm

Our analysis of locality shows that differently transformed loops can differ in locality only if they have different localized vector spaces. That is, all transformations that generate code with the same localized vector space can be put in an equivalence class and need not be examined individually. The only feature of interest is the innermost tile; the outer loops can be executed in any legal order or orientation. Similarly, reordering or skewing the tiled loops themselves does not affect the vector space and thus need not be considered.

From the reuse analysis, we can identify a subspace that is desirable to make into the innermost tile. The question of whether there exists a unimodular transformation that is legal and creates such an innermost subspace is a difficult one. An existing algorithm that attempts to find such a transform is exponential in the number of loops [15]. The general question of finding a legal transformation that minimizes the number of memory accesses as determined by the intersection of the localized and reused vector spaces is even harder.

Although the problem is theoretically difficult, loop nests found in practice are generally simple. Using characteristics of programs as a guide, we simplify this problem by (1) reducing the set of equivalent classes, and (2) using a heuristic algorithm for finding transforms.

5.1 Loops Carrying Reuse

Although reuse vector spaces can theoretically be spanned by arbitrary vectors, in practice they are typically spanned by a subset of the elementary basis of the iteration space, that is, by a a subset of the loop axes. In the code below, the array A has self-temporal reuse in the vector space spanned by $(0,0,1)$; we say that the loop I_3 carries reuse.

```
for $I_1$
  for $I_2$
    for $I_3$
      A[$I_1$,$I_2$] := B[$I_1 + I_2$,$I_3$];
```

On the other hand, the B array has a self-temporal reuse in the $(1,-1,0)$ direction of the iteration space, which does not correspond to either loop I_1 or I_2 but rather a combination of them. This latter situation is not as common.

Instead of using an arbitrary reuse vector space directly to guide the transformation process, we use its smallest enclosing space spanned by the elementary vectors of the iteration space. For example, the reuse vector space of B[I_1+I_2, I_3] is spanned by $(1,0,0)$ and $(0,1,0)$, the directions of loops I_1 and I_2 respectively. If we succeed in making the innermost tile include both of these directions, we will exploit the self-temporal reuse of the reference. Informally, this procedure reduces the arbitrary vector spaces to a set of loops that carry reuse.

With this simplification, a loop in the source program either carries reuse or it does not. We partition all transformations into equivalence classes according to the set of source loop directions included in the localized vector space. For example, both loops in Figure 2(a) carry reuse. Transformations are classified depending on whether the transformed innermost tile contains only the first loop, the second or both.

5.2 An Algorithm to Improve Locality

We first prune the search by finding those loop directions that need not or cannot be included in the localized vector space. These loops include those that carry no reuse and can be placed outermost legally, and those that carry reuse but must be placed outermost due to legality reasons. For example, if a three-deep loop has dependences

$$\{(1, `\pm', `\pm'), (0, 1, `\pm'), (0, 0, `+')\}$$

and carry locality in all three loops, there is no need to try all the different transformations when clearly only one loop ordering is legal.

On the remaining loops $\mathcal{I}$, we examine every subset I of $\mathcal{I}$ that contains at least some reuse. For each set I, we try to find a legal transformation such that the loops in I are tiled innermost. Among all the legal transformations, we select the one with the minimal memory accesses per iteration. Because the power set of $\mathcal{I}$ is explored, this algorithm is exponential in the number of loops in $\mathcal{I}$. However, $\mathcal{I}$, a subset of the original loops, is typically small in practice.

There are two steps in finding a transformation that makes loops I innermost. We first attempt to order the outer loops—the loops in $\mathcal{I}$ but not in I. Any transformation applied to the loops in $\mathcal{I} - I$ that result in these loops being outermost and no dependences being violated by these loops is sufficient [16]. If that step succeeds, then we attempt to tile the I loops innermost, which means finding a transformation that turns these loops into a fully permutable loop nest, given the outer nest. Solving these problems exactly is still exponential in the number of dependences. We have developed a heuristic compound transformation, known as the SRP transform, which is useful in both steps of the transformation algorithm.

The SRP transformation attempts to make a set of loops fully permutable by applying combinations of permutation, skewing and reversal [16]. If it cannot place all loops in a single fully permutable nest, it simply finds the outermost nest, and returns all remaining loops and dependences left to be made lexicographically positive. The algorithm is based upon the observations in Theorem 5.1 and Corollary 5.2.

Theorem 5.1 *Let $N = \{I_1, \ldots, I_n\}$ be a loop nest with lexicographically positive dependences $\vec{d} \in D$, and $D^i = \{\vec{d} \in D | (d_1, \ldots, d_{i-1}) \not\succ \vec{0}\}$. Loop I_j can be made into a fully permutable nest with loop I_i, where $i < j$, via reversal and/or skewing, if*

$$\forall \vec{d} \in D^i : (d_j^{\min} \neq -\infty \wedge (d_j^{\min} < 0 \rightarrow d_i^{\min} > 0)), \textit{ or}$$

$$\forall \vec{d} \in D^i : (d_j^{\max} \neq \infty \wedge (d_j^{\max} > 0 \rightarrow d_i^{\min} > 0)).$$

Proof: All dependence vectors for which $(d_1, \ldots, d_{i-1}) \succ \vec{0}$ do not prevent loops I_i and I_j from being fully permutable and can be ignored. If

$$\forall \vec{d} \in D^i : (d_j^{\min} \neq -\infty \wedge (d_j^{\min} < 0 \rightarrow d_i^{\min} > 0))$$

then we can skew loop I_j by a factor of f with respect to loop I_i where

$$f \geq \max_{\{\vec{d} | \vec{d} \in D^i \wedge d_i^{\min} \neq 0\}} \lceil -d_j^{\min} / d_i^{\min} \rceil$$

to make loop I_j fully permutable with loop I_i. If instead the condition

$$\forall \vec{d} \in D^i : (d_j^{\max} \neq \infty \wedge (d_j^{\max} > 0 \rightarrow d_i^{\min} > 0)).$$

holds, then we can reverse loop I_j and proceed as above. □

Corollary 5.2 *If loop I_k has dependences such that $\exists \vec{d} \in D : d_k^{\min} = -\infty$ and $\exists \vec{d} \in D : d_k^{\max} = \infty$ then the outermost fully permutable nest consists only of a combination of loops not including I_k.*

The SRP algorithm takes as input the loops N that have not been placed outside this nest, and the set of dependences D that have not been satisfied by loops outside this loop nest. It first removes from N those *serializing loops* as defined by the I_ks of Corollary 5.2. It then uses an iterative step to build up the fully permutable loop nest F. In each step, it tries to find a loop from the remaining loops in N that can be made fully permutable with F via possibly multiple applications of Theorem 5.1. If it succeeds in finding such a loop, it permutes the loop to be next outermost in the fully permutable nest, adding the loop to F and removing it from N. Then it repeats, searching through the remaining loops in N for another loop to place in F. This algorithm is known as SRP because the unimodular transformation it performs can be expressed as the product of a skew transformation (S), a reversal transformation (R) and a permutation transformation (P).

We use SRP in both steps of finding a transformation that makes a loop nest I the innermost tile. We first apply SRP iteratively to those loops not in I. Each step through the SRP algorithm attempts to find the next outermost fully permutable loop nest, returning all remaining loops that cannot be made fully permutable and returning the unsatisfied dependences. We repeatedly call SRP on the remaining loops until (1) SRP fails to find a single loop to place outermost, in which case the algorithm fails to find a legal ordering for this target innermost tile I, or (2) there are no remaining loops, in which case the algorithm succeeds. If this step succeeds, we then call SRP with loops I, and all remaining dependences to be satisfied. In this case, we succeed only if SRP makes all the loops fully permutable.

Let us illustrate SRP algorithm using the example in Figure 2. Suppose we are trying to tile loops I_1 and I_2. First an outer loop must be chosen. I_1 can be the outer loop, because its dependence components are all non-negative. Now loop I_2 has a dependence component that is negative, but it can be made non-negative by skewing with respect to I_1 (Figure 2D). Loop I_2 is now placed in the same fully permutable nest as I_1; the loop nest is tilable (Figure 2(e)).

In general, SRP can be applied to loop nests of arbitrarily depth where the dependences can include distances and directions. In the important special case where all the dependences in the loop nest to be ordered are lexicographically positive distance vectors, the algorithm can place all the loops into a single fully permutable loop nest. The algorithm is $O(n^2 d)$, where n is the loop nest depth and d is the number of dependence vectors. With the extension of a simple 2D time-cone solver[16], it becomes $O(n^3 d)$ but can find a transformation that makes any two loops fully permutable, and therefore tilable, if some such transformation exists.

If there is little reuse, or if data dependences constrain the legal ordering possibilities, the algorithm is fast since $\mathcal{I}$ is small. The algorithm is only slow when there are many carrying reuse loops and few dependences. This algorithm can be further improved by ordering the search through the power set of $\mathcal{I}$ using a branch and bound approach.

6 Tiling Experiments

We have implemented the algorithm described in this paper in our SUIF compiler. The compiler currently generates working tiled code for one and multiple processors of the SGI 4D/380. However, the scalar optimizer in our compiler has not yet been completed, and the numbers obtained with our generated code would not reflect the true effect of tiling when the code is optimized. Fortunately, our SUIF compiler also includes a C backend; we can use the compiler to generate restructured C code, which can then be compiled by a commercial scalar optimizing compiler.

The numbers reported below are generated as follows. We used the compiler to generate tiled C code for a single processor. We then performed, by hand, optimizations such as register allocation of array elements[5], moving loop-invariant address calculation code out of the innermost loop, and unrolling the innermost loop. We then compiled the code using the SGI's optimizing compiler. To run the code on multiple processors, we adopt the model of executing the tiles in a DO-ACROSS manner[16]. This code has the same structure as the sequential code. We needed to add only a few lines to the sequential code to create multiple threads, and initialize, check and increment a small number of counters within the outer loop.

6.1 LU Decomposition

The original code for LU-decomposition is:

```
for $I_1$ := 1 to $n$ do
  for $I_2$ := $I_1$+1 to $n$ do
```

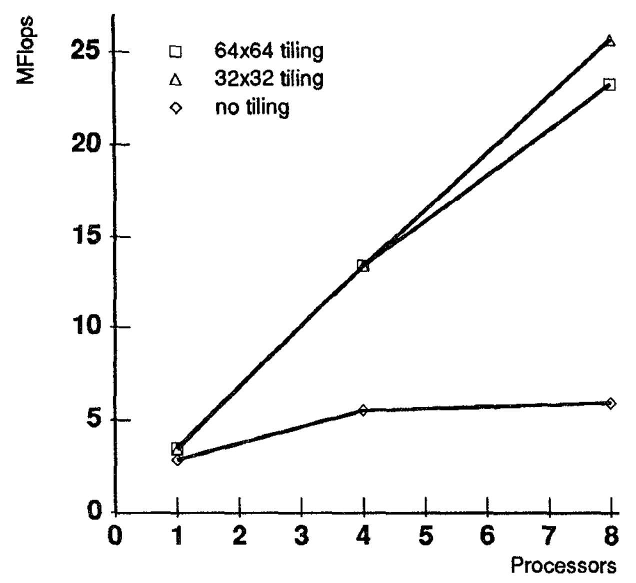


Figure 5: Performance of 500 x 500 double precision LU factorization without pivoting on the SGI 4D/380. No register tiling was performed.

```
A[$I_2$,$I_1$] /= A[$I_1$,$I_1$];
for $I_3$ := $I_1$+1 to $n$ do
  A[$I_2$,$I_3$] -= A[$I_2$,$I_1$]*A[$I_1$,$I_3$];
```

For comparison, we measured the performance of the original untiled LU code on one, four and eight processors. For this code, we allocated the element A[I_2, I_1] to a register in the duration of the innermost loop. We parallelized the middle loop, and ran it self-scheduled, with 15 iterations per task. (The results for larger and smaller granularities are virtually identical.) Figure 5 shows that we only get a speedup of approximately 2 on eight processors, even though there is plenty of available parallelism. This is because the memory bandwidth of the machine is not sufficient to support eight processors that are each reusing so little of the data in their respective caches.

This LU loop nest has a reuse vector space that spans all three loops, so our compiler tiles all the loops. The LU code is not perfectly nested, since the division is outside of the innermost loop. The generated code is thus more complicated:

```
for $II_2$ = 1 to $n$ by $s$ do
 for $II_3$ = 1 to $n$ by $s$ do
  for $I_1$ = 1 to $n$ do
   for $I_2$ = max($I_1+1$,$II_2$) to
            min($n$,$II_2+s-1$) do
     if $II_3 \le I_1+1 \le II_3+s-1$ then
       A[$I_2$,$I_1$] /= A[$I_1$,$I_1$];
     for $I_3$ = max($I_1+1$,$II_3$) to
               min($n$,$II_3+s-1$) do
       A[$I_2$,$I_3$] -= A[$I_2$,$I_1$]*A[$I_1$,$I_3$];
```

As with matrix multiplication, while tiling for cache reuse is important for one processor, it is crucial for multiple processors. Tiling improves the uniprocessor execution by about 20%; more significantly, the speedup is much closer to linear in the number of processors when tiling is used. We ran experiments with tile sizes 32 x 32 and 64 x 64. The results coincide with our analysis that the marginal performance gain due to increases in the tile size is low. In this example, a 32 x 32 tile, which holds 1/4 the data of a 64 x 64 tile, performed virtually as well as the larger on a single processor. It actually outperformed the larger tile in the eight processor experiment, mostly due to load balancing.

6.2 SOR

We also ran experiments with the SOR code in Figure 7(a), which is a two-dimensional equivalent of the example in Figure 2(a). Figure 6 shows the performance results for three versions of this nest, where t is 30 and $n+1$, the size of the matrix, is 500. None of the original loops is parallelizable. The first verson, labeled "DOALL", is transformed via wavefronting so that the middle loop is a DOALL loop [16] (Figure 7(b)). This transformation unfortunately destroys the original locality within the code. Thus, performance is abysmal on a single processor, and speedup for multiple processors is equally abysmal even though again there is plenty of available parallelism in the I_2' loop.

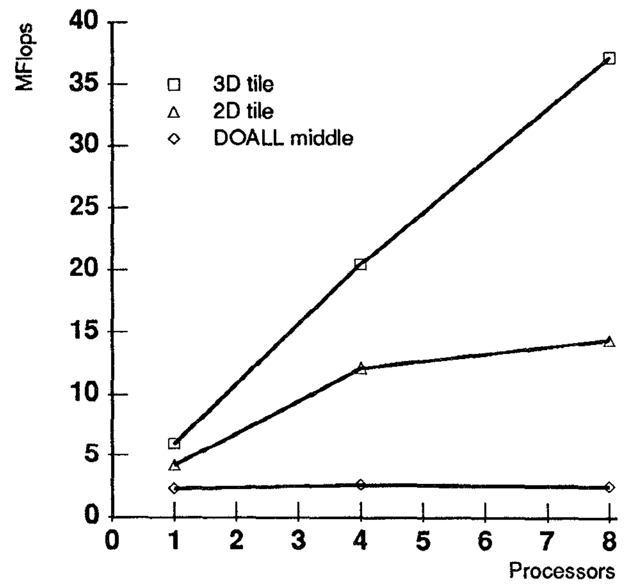


Figure 6: Behavior of 30 iterations of a 500 x 500 double precision SOR step on the SGI 4D/380. The tile sizes are 64 x 64 iterations. No register tiling was performed.

The code for the second version, labeled "2D tile", is shown in Figure 7(c). This version tiles only the inner-

(a): SOR nest

```
for $I_1$ := 1 to $t$ do
  for $I_2$ := 1 to $n-1$ do
    for $I_3$ := 1 to $n-1$ do
      A[$I_2, I_3$] := 0.2*(A[$I_2, I_3$] + A[$I_2+1, I_3$] + A[$I_2-1, I_3$] + A[$I_2, I_3+1$] + A[$I_2, I_3-1$]);
```

(b): "DOALL" SOR nest

```
for $I'_1$ := 5 to $2N-2+3t$ do
  doall $I'_2$ := max(1,$(I'_1-t-2N+2)/2$) to min($t$,$(I'_1-3)/2$) do
    for $I'_3$ := max($1+I'_2, I'_1-2I'_2-n+1$) to min($n-1+I'_2, I'_1-1-2I'_2$) do
      A[$I'_3-I'_2, I'_1-2I'_2-I'_3$] := 0.2*(A[$I'_3-I'_2, I'_1-2I'_2-I'_3$] + A[$I'_3-I'_2+1, I'_1-2I'_2-I'_3$]
            + A[$I'_3-I'_2-1, I'_1-2I'_2-I'_3$] + A[$I'_3-I'_2, I'_1-2I'_2-I'_3+1$] + A[$I'_3-I'_2, I'_1-2I'_2-I'_3-1$]);
```

(c): "2-D Tile" SOR nest

```
for $I'_1$ := 1 to $t$ do
  for $II'_3$ := 1 to $n-1$ by $s$ do
    for $I'_2$ := 1 to $n-1$ do
      for$I'_3$ := $II'_3$ to min($n-1, II_3+s-1$) do
        A[$I'_2, I'_3$] := 0.2*(A[$I'_2$][$I'_3$]+A[$I'_2+1, I'_3$]+A[$I'_2-1, I'_3$]+A[$I'_2, I'_3+1$]+A[$I'_2, I'_3-1$]);
```

(d): "3-D Tile" SOR nest

```
for $II'_2$ := 2 to $n-1+t$ by $s$ do
  for $II'_3$ := 2 to $n-1+t$ by $s$ do
    for $I'_1$ := 1 to $t$ do
      for $I'_2$ := $II'_2$ to min($n-1+t, II'_2+s-1$) do
        for $I'_3$ := $II'_3$ to min($n-1+t, II'_3+s-1$) do
          A[$I'_2, I'_3$] := 0.2*(A[$I'_2-I'_1$][$I'_3-I'_1$] + A[$I'_2-I'_1+1, I'_3-I'_1$]
              + A[$I'_2-I'_1-1, I'_3-I'_1$] + A[$I'_2-I'_1, I'_3-I'_1+1$] + A[$I'_2-I'_1, I'_3-I'_1-1$]);
```

Figure 7: Code for the different SOR versions.

most two loops. The uniprocessor performance is significantly better than even the eight processor wavefronted performance. Thus although wavefronting may increase parallelism, it may reduce locality so much that it is better to use only one processor. However, the speedup of the 2D tile method is still limited because self-temporal reuse is not being exploited.

To exploit all the available reuse, all three loops must be included in the innermost tile. The SRP algorithm first skews I_2 and I_3 with respect to I_1 to make tiling legal, and then tiles to produce the code in Figure 7(d). This "3D tile" version of the loop nest is best for a single processor, and also has the best speedup in the multiprocessor version.

7 Conclusions

In this paper, we propose a complete approach to the problem of improving the cache performance for loop nests. The approach is to first transform the code via interchange, reversal, skewing and tiling, then determine the tile size, taking into account data conflicts due to the set associativity of the cache [12]. The loop transformation algorithm is based on two concepts: a mathematical formulation of reuse and locality, and a matrix-based loop transformation theory.

While previous work on evaluating locality estimates the number of memory accesses directly for a given transformed code, we break the evaluation into three parts. We use a *reuse vector space* to capture the inherent reuse within a loop nest; we use a *localized vector space* to capture a compound transform's potential to exploit locality; finally we evaluate the locality of a transformed code by intersecting the reuse vector space with the localized vector space.

There are four reuse vector spaces: self-temporal, self-spatial, group-temporal, and group-spatial. These reuse vector spaces need to be calculated only once for a given loop nest. We show that while unimodular transformations can alter the orientation of the localized vector space, tiling can increase the dimensionality of this space.

The reuse and localized vector spaces can be used to prune the search for the best compound transformation, and not just for evaluating the locality of a given code. First, all transforms with identical localized vector space are equivalent with respect to locality. In addition, transforms with different localized vector spaces may also be equivalent if the intersection between the localized and the reuse vector spaces is identical. A loop transformation algorithm need only to compare between transformations that give different localized and reuse vector space intersections.

Unlike the stepwise transformation approach used in existing compilers, our loop transformer solves for the *compound* transformation directly. This is made possible by our theory that unifies loop interchanges, skews and reversals as unimodular matrix transformations on dependence vectors with either direction or distance components. The algorithm extracts the dependence vectors, determines the best compound transform using locality objectives to prune the search, then transforms the loops and their loop bounds once and for all. This theory makes the implementation of the algorithm simple and straightforward.

References

[1] W. Abu-Sufah. *Improving the Performance of Virtual Memory Computers*. PhD thesis, University of Illinois at Urbana-Champaign, Nov 1978.

[2] U. Banerjee. Data dependence in ordinary programs. Technical Report 76-837, University of Illinios at Urbana-Champaign, Nov 1976.

[3] U. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic, 1988.

[4] U. Banerjee. Unimodular transformations of double loops. In *3rd Workshop on Languages and Compilers for Parallel Computing*, Aug 1990.

[5] D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, June 1990.

[6] J. Dongarra, J. Du Croz, S. Hammarling, and I. Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, pages 1–17, March 1990.

[7] K. Gallivan, W. Jalby, U. Meier, and A. Sameh. The impact of hierarchical memory systems on linear algebra algorithm design. Technical report, University of Illinios, 1987.

[8] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformation. *Journal of Parallel and Distributed Computing*, 5:587–616, 1988.

[9] G. H. Golub and C. F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, 1989.

[10] F. Irigoin and R. Triolet. Computing dependence direction vectors and dependence cones. Technical Report E94, Centre D'Automatique et Informatique, 1988.

[11] F. Irigoin and R. Triolet. Supernode partitioning. In *Proc. 15th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, January 1988.

[12] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991.

[13] A. C. McKeller and E. G. Coffman. The organization of matrices and matrix operations in a paged multiprogramming environment. *CACM*, 12(3):153–165, 1969.

[14] A. Porterfield. *Software Methods for Improvement of Cache Performance on Supercomputer Applications*. PhD thesis, Rice University, May 1989.

[15] R. Schreiber and J. Dongarra. Automatic blocking of nested loops. 1990.

[16] M. E. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, July 1991.

[17] M. J. Wolfe. Techniques for improving the inherent parallelism in programs. Technical Report UIUCDCS-R-78-929, University of Illinois, 1978.

[18] M. J. Wolfe. More iteration space tiling. In *Supercomputing '89*, Nov 1989.