Compiler Optimizations for Improving Data Locality

Steve Carr
carr@cs.mtu.edu
Department of Computer Science
Michigan Technological University

Kathryn S. McKinley
mckinley@cs.umass.edu
Department of Computer Science
University of Massachusetts

Chau-Wen Tseng
tseng@cs.stanford.edu
Computer Systems Laboratory
Stanford University

Abstract

In the past decade, processor speed has become significantly faster than memory speed. Small, fast cache memories are designed to overcome this discrepancy, but they are only effective when programs exhibit *data locality*. In this paper, we present compiler optimizations to improve data locality based on a simple yet accurate cost model. The model computes both *temporal* and *spatial* reuse of cache lines to find desirable loop organizations. The cost model drives the application of compound transformations consisting of loop permutation, loop fusion, loop distribution, and loop reversal. We demonstrate that these program transformations are useful for optimizing many programs.

To validate our optimization strategy, we implemented our algorithms and ran experiments on a large collection of scientific programs and kernels. Experiments with kernels illustrate that our model and algorithm can select and achieve the best performance. For over thirty complete applications, we executed the original and transformed versions and simulated cache hit rates. We collected statistics about the inherent characteristics of these programs and our ability to improve their data locality. To our knowledge, these studies are the first of such breadth and depth. We found performance improvements were difficult to achieve because benchmark programs typically have high hit rates even for small data caches; however, our optimizations significantly improved several programs.

1 Introduction

Because processor speed is increasing at a much faster rate than memory speed, computer architects have turned increasingly to the use of memory hierarchies with one or more levels of cache memory. Caches take advantage of *data locality* in programs. Data locality is the property that references to the same memory location or adjacent locations are reused within a short period of time.

Caches also have an impact on programming; programmers substantially enhance performance by using a style that ensures more memory references are handled by the cache. Scientific programmers expend considerable effort at improving locality by structuring loops so that the innermost loop iterates over the elements of a column, which are stored consecutively in Fortran. This task is time consuming, tedious, and error-prone. Instead, achieving good data locality should be the responsibility of the compiler. By placing the burden on the compiler, programmers can get good uniprocessor performance even if they originally wrote their program for a vector or parallel machine. In addition, programs will be more portable because programmers will be able to achieve good performance without making machine-dependent source-level transformations.

Steve Carr was supported by NSF grant CCR-9409341. Chau-Wen Tseng was supported in part by an NSF CISE Postdoctoral Fellowship in Experimental Science.

To appear in *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, San Jose, CA, October 1994.

1.1 Optimization Framework

Based on our experiments and experiences, we believe that compiler optimizations to improve data locality should proceed in the following order:

1. Improve order of memory accesses to exploit all levels of the memory hierarchy through loop permutation, fusion, distribution, skewing, and reversal. This process is mostly machine-independent, and requires knowledge only of the cache line size.
2. Fully utilize the cache through *tiling*, a combination of strip-mining and loop permutation [IT88]. Knowledge of the cache size, associativity, and replacement policy is essential. Higher degrees of tiling can be applied to exploit multi-level caches, the TLB, etc.
3. Promote register reuse through *unroll-and-jam* (also known as register tiling) and *scalar replacement* [CCK90]. The number and type of registers available are required to determine the degree of unroll-and-jam and the number of array references to replace with scalars.

In this paper, we concentrate on the first step. Our algorithms are complementary to and in fact, improve the effectiveness of optimizations performed in the latter two steps [Car92]. However, the other steps and interactions between steps are beyond the scope of this paper.

1.2 Overview

We present a compiler strategy based on an effective, yet simple, model for estimating the cost of executing a given loop nest in terms of the number of cache line references. This paper extends previous work [KM92] with a slightly more accurate memory model. We use the model to derive a loop structure which results in the fewest accesses to main memory. The loop structure is achieved through an algorithm that uses compound loop transformations. The compound loop transformations are permutation, fusion, distribution, and reversal. The algorithm is unique to this paper and is implemented in a source-to-source Fortran 77 translator.

We present extensive empirical results for kernels and benchmark programs that validate the effectiveness of our optimization strategy; they reveal programmers often use programming styles with good locality. We measure both inherent data locality characteristics of scientific programs and our ability to improve data locality. When the cache miss rate for a program is non-negligible, we show there are often opportunities to improve data locality. Our optimization algorithm takes advantage of these opportunities and consequently improves performance. As expected, loop permutation plays the key role; however, loop fusion and distribution may also produce significant improvements. Our algorithms never found an opportunity where loop reversal could improve locality.

2 Related Work

Abu-Sufah first discussed applying compiler transformations based on data dependence (*e.g.,* loop interchange, fusion, distribution, and tiling) to improve paging [AS79]. In this paper, we extend and validate recent research to integrate optimizations for parallelism and memory [KM92]. We extend their original cost model to capture more types of reuse. The only transformation they perform is loop permutation, whereas we integrate permutation, fusion, distribution, and reversal into a comprehensive approach. Our extensive experimental results are unique to this paper. We measure both the effectiveness of our approach and unlike other optimization studies, the inherent data locality characteristics of programs and our ability to exploit them.

Our approach has several advantages over previous research. It is applicable to a wider range of programs because we do not require perfect nests or nests that can be made perfect with conditionals [FST91, GJG88, LP92, WL91]. It is quicker, both in the expected and worse case. Previous work generates all loop permutations [FST91, GJG88] or unimodular transformations (a combination of permutation, skewing, and reversal) [LP92, WL91], evaluates the locality of all legal permutations, and then picks the best. This process requires the evaluation of up to $n!$ loop permutations (though n is typically small). In comparison, our approach only performs one evaluation step because it directly determines the best loop permutation.

Wolf and Lam use unimodular transformations and tiling with estimates of temporal and spatial reuse to improve data locality [WL91]. Their memory model is potentially more precise than ours because it directly calculates reuse across outer loops; however, it may be less precise because it ignores loop bounds even when they are known constants. Wolf and Lam's evaluation is performed on the Perfect Benchmarks and routines in *Dnasa7* in the SPEC Benchmarks, a subset of our test suite. It is difficult to directly compare our experiments because their cache optimization results include tiling and scalar replacement and are executed on a different processor. We improve a few more programs/routines than they do, but their cache optimizations degrade 6 programs/routines, in one case by 20% [Wol92]. We degrade one program by 2%, *Applu* from the NAS Benchmarks.

In Wolf and Lam's experiments, skewing was never needed and reversal was seldom applied [Wol92]. We therefore chose not to include skewing, even though it is implemented in our system [KMT93] and our model can drive it. We did integrate reversal, but it did not help to improve locality.

The exhaustive approach taken by previous researchers [FST91, GJG88, LP92, WL91] is not practical when including transformations which create and combine loop nests (*e.g.,* fusion, distribution). Fusion for improving reuse is by itself NP-hard [KM93]. By driving heuristics with a cache model, our algorithms are efficient and usually find the best transformations for data locality using permutation, fusion and distribution.

3 Background

In this section, we characterize data reuse and present an effective data locality cost model.

3.1 Data Dependence

We assume the reader is familiar with concept of data dependence [KKP+81, GKT91]. $\vec{\delta} = \{\delta_1 \ldots \delta_k\}$ is a hybrid distance/direction vector with the most precise information derivable. It represents a data dependence between two array references, corresponding left to right from the outermost loop to innermost loop enclosing the references. Data dependences are loop-independent if the accesses to the same memory location occur in the same loop iteration; they are loop-carried if the accesses occur on different loop iterations.

3.2 Sources of Data Reuse

The two sources of data reuse are *temporal* reuse, multiple accesses to the same memory location, and *spatial* reuse, accesses to nearby memory locations that share a cache line or a block of memory at some level of the memory hierarchy (Unit-stride access is the most common type of spatial locality). Temporal and spatial reuse may result from *self-reuse* from a single array reference or *group-reuse* from multiple references. Without loss of generality, in this paper we assume Fortran's column-major storage.

To simplify analysis, we concentrate on reuse that occurs between small numbers of inner loop iterations. Our memory model assumes there will be no conflict or capacity cache misses in one iteration of the innermost loop.[1] We use the algorithms *RefGroup*, *RefCost* and *LoopCost* to determine the total number of cache lines accessed when a candidate loop l is placed in the innermost loop position. The result reveals the relative amounts of reuse between loops in the same nest and across disjoint nests; it also drives permutation, fusion, distribution, and reversal to improve data locality.

3.3 Reference Groups

Our cost model first applies algorithm *RefGroup* to calculate group-reuse. Two references are in the same *reference group* if they exhibit group-temporal or group-spatial reuse, *i.e.,* they access the same cache line on the same or different iterations of an inner loop. This formulation is more general than previous work [KM92], but slightly more restrictive than *uniformly generated references* [GJG88].

RefGroup: Two references Ref_1 and Ref_2 belong to the same reference group with respect to loop l if:

1. $\exists$ $Ref_1 \; \vec{\delta} \; Ref_2$, and
   (a) $\vec{\delta}$ is a loop-independent dependence, or
   (b) δ_l is a small constant d ($|d| \leq 2$) and all other entries are zero,
2. or, Ref_1 and Ref_2 refer to the same array and differ by at most d' in the first subscript dimension, where d' is less than or equal to the cache line size in terms of array elements. All other subscripts must be identical.

Condition 1 accounts for group-temporal reuse and condition 2 detects most forms of group-spatial reuse.

3.4 Loop Cost in Terms of Cache Lines

Once we account for group-reuse, we can calculate the reuse carried by each loop using the functions *RefCost* and *LoopCost* in Figure 1. To determine the cost in cache lines of a reference group, we select an arbitrary array reference with the deepest nesting from each group. Each loop l with *trip* iterations in the nest is considered as a candidate for the innermost position. Let *cls* be the cache line size in data items and *stride* be the step size of l multiplied by the coefficient of the loop index variable.

In Figure 1, *RefCost* calculates locality for l, *i.e.,* the number of cache lines l uses: 1 for loop-invariant references, *trip/(cls/stride)* for consecutive references, or *trip* for non-consecutive references. *LoopCost* then calculates the total number of cache lines accessed by all references when l is the innermost loop. It simply sums *RefCost* for all reference groups, then multiplies the result by the trip counts

[1] Lam *et al.* confirm this assumption [LRW91].

Figure 1: LoopCost Algorithm

$$
\begin{array}{lrcl}
\text{INPUT:} & \mathcal{L} & = & \{l_1, \ldots, l_n\} \text{ a loop nest with headers } lb,\, ub,\, \textit{step} \\
 & \mathcal{R} & = & \{\, \textit{Ref}_1, \ldots, \textit{Ref}_m\} \text{ representatives from each reference group} \\
 & trip_l & = & (ub_l - lb_l + \textit{step}_l)/\textit{step}_l \\
 & \textit{cls} & = & \text{the cache line size,} \\
 & \textit{coeff}(i_l, f) & = & \text{the coefficient of the index variable } i_l \text{ in the subscript } f \text{ (it may be zero)} \\
 & \textit{stride} & = & |\ \text{step}_l * \textit{coeff}(f_1, i_l)| \\
\text{OUTPUT:} & \textit{LoopCost}(l) & = & \text{number of cache lines accessed with } l \text{ as innermost loop} \\
\text{ALGORITHM:} & \textbf{LoopCost}(l) & = & \displaystyle\sum_{k=1}^{m} \Big(\textbf{RefCost}(\textit{Ref}_k(f_1(i_1, \ldots, i_n), \ldots, f_j(i_1, \ldots, i_n)), l)\Big) \prod_{h \neq l} \textit{trip}_h
\end{array}
$$

$$
\textbf{RefCost}(\textit{Ref}_k, l) = \begin{cases}
1 & \textit{if } (\textit{coeff}(f_1, i_l) = 0) \wedge \ldots \wedge (\textit{coeff}(f_j, i_l) = 0) \quad \textbf{loop invariant} \\
\dfrac{\text{trip}_l}{\textit{cls / stride}} & \textit{if } ((\textit{stride} < \textit{cls}) \wedge (\textit{coeff}(f_2, i_l) = 0) \wedge \ldots \wedge (\textit{coeff}(f_j, i_l) = 0) \quad \textbf{consecutive} \\
\textit{trip}_l & \textit{otherwise} \quad \textbf{no reuse}
\end{cases}
$$

of all the remaining loops. *RefCost* and *LoopCost* appear in Figure 1. This method evaluates imperfectly nested loops, complicated subscript expressions and nests with symbolic bounds [McK92].

4 Compound Loop Transformations

In this section, we show how the cost model guides loop permutation, fusion, distribution, and reversal. Each subsection describes tests based on the cost model to determine when individual transformations are profitable. Using these components, Section 4.5 presents *Compound*, an algorithm for discovering and applying legal compound loop nest transformations that attempt to minimize the number of cache lines accessed. All of these transformations are implemented.

4.1 Loop Permutation

To determine the loop permutation which accesses the fewest cache lines, we rely on the following observation.

> *If loop l promotes more reuse than loop l' when both are considered as innermost loops, l will promote more reuse than l' at any outer loop position.*

We therefore simply rank the loops using *LoopCost*, ordering the loops from outermost to innermost ($l_1 \ldots l_n$) so that *LoopCost*(l_{i-1}) $\geq$ *LoopCost*(l_i). We call this permutation of the nest with the least cost *memory order*. If the bounds are symbolic, we compare the dominating terms.

We define the algorithm *Permute* to achieve memory order when possible on perfect nests.[2] To determine if the order is a legal one, we permute the corresponding entries in the distance/direction vector. If the result is lexicographically positive the permutation is legal and we transform the nest. If a legal permutation exists which positions the loop with the most reuse innermost, the algorithm is guaranteed to find it. If the desired inner loop cannot be obtained, the next most desirable inner loop is positioned innermost if possible, and so on. Because most data reuse occurs on the innermost loop, positioning it correctly yields the best data locality.

Complexity. When memory order is legal, as it is in 80% of loops in our test suite, *Permute* simply sorts the loops in $O(n \log n)$ time. If it is not legal, *Permute* selects a legal permutation as close to memory order as possible, taking worst case $n(n-1)$ time [KM92]. These steps are inexpensive; evaluating the locality of the nest is the most expensive step. Our algorithm computes the best permutation with one evaluation step for each loop in the nest. The complexity of this step is therefore $O(n)$ time, where n is in the number of loops in the nest.

4.1.1 Example: Matrix Multiplication

In Figure 2, algorithm *RefGroup* for matrix multiply puts the two references to C(I,J) in the same reference group and A(I,K) and B(K,J) in separate groups for all loops. Algorithm *MemoryOrder* uses *LoopCost* to select JKI as memory order; A(I,K) and C(I,J) exhibit spatial locality and B(K,J) exhibits loop invariant temporal locality, resulting in the fewest cache line accesses.

To validate our cost model, we gathered results for all possible permutations, ranking them left to right from the least to the highest cost (JKI, KJI, JIK, IJK, KIJ, IKJ) in Figure 2. Consistent with our model, choosing I as the inner loop results in the best execution time. Changing the inner loop has a dramatic effect on performance. The impact is greater on the 512 $\times$ 512 versus the 300 $\times$ 300 matrices because a larger portion of the working set stays in the cache. Execution times vary by significant factors of up to 3.7 on the Sparc2, 6.2 on the i860, and 23.9 on the RS/6000. The entire ranking accurately predicts relative performance.

We performed this type of comparison on several more kernels and a small program with the same result: memory order always resulted in the best performance.

4.2 Loop Reversal

Loop reversal reverses the order in which the iterations of a loop nest execute and is legal if dependences remain carried on outer loops. Reversal does not change the pattern of reuse, but it is an *enabler*; *i.e.,* it may enable permutation to achieve better locality. We extend *Permute* to perform reversal as follows. If memory order is not legal, *Permute* places outer loops in position first, building up lexicographically positive dependence vectors [KM92]. If *Permute* cannot legally position a loop in a desired position, *Permute* tests if reversal is legal and enables the loop to be put in the position. Reversal did not improved locality in our experiments, therefore we will not discuss it further.

4.3 Loop Fusion

Loop fusion takes multiple loop nests and combines their bodies into one loop nest. It is legal only if no data dependences are reversed [War84]. As an example of its effect, consider the *scalarization* into Fortran 77 in Figure 3(b) of the Fortran 90 code fragment for performing ADI integration in Figure 3(a). The Fortran 90 code

[2] In Section 4.5, we perform imperfect interchanges with distribution.

Figure 2: Matrix Multiply

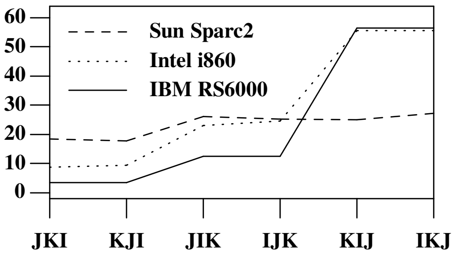
```
{ JKI ordering }
DO J = 1, N
    DO K = 1, N
        DO I = 1, N
            C(I,J) = C(I,J) + A(I,K) * B(K,J)
```

LoopCost (with $cls = 4$)

Refs	J	K	I
C(I,J)	$n * n^2$	$1 * n^2$	$\frac{1}{4}n * n^2$
A(I,K)	$1 * n^2$	$n * n^2$	$\frac{1}{4}n * n^2$
B(K,J)	$n * n^2$	$\frac{1}{4}n * n^2$	$1 * n^2$
total	$2n^3 + n^2$	$\frac{5}{4}n^3 + n^2$	$\frac{1}{2}n^3 + n^2$

Execution Times (in seconds)

300 x 300



512 x 512

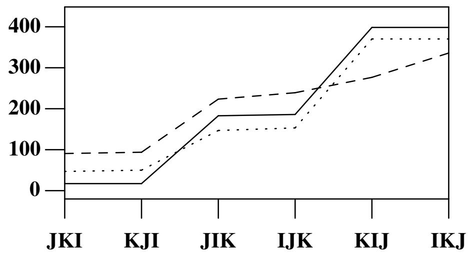


Figure 3: Loop Fusion

(a) *Sample Fortran 90 loops for ADI Integration*

```
     DO I = 2, N
S1     X(I,1:N) = X(I,1:N) - X(I-1,1:N)*A(I,1:N)/B(I-1,1:N)
S2     B(I,1:N) = B(I,1:N) - A(I,1:N)*A(I,1:N)/B(I-1,1:N)
```

(b) ⇓ *Translation to Fortran 77* ⇓

```
DO I = 2, N
  DO K = 1, N
    X(I,K) = X(I,K) - X(I-1,K)*A(I,K)/B(I-1,K)
  DO K = 1, N
    B(I,K) = B(I,K) - A(I,K)*A(I,K)/B(I-1,K)
```

(c) ⇓ *Loop Fusion & Interchange* ⇓

```
DO K = 1, N
  DO I = 2, N
    X(I,K) = X(I,K) - X(I-1,K)*A(I,K)/B(I-1,K)
    B(I,K) = B(I,K) - A(I,K)*A(I,K)/B(I-1,K)
```

LoopCost (with $cls = 4.$)

RefGroup	K	I
X(I,K)	$n * n$	$\frac{1}{4}n * n$
A(I,K)	$n * n$	$\frac{1}{4}n * n$
B(I,K)	$n * n$	$\frac{1}{4}n * n$
total	$3 * n^2$	$\frac{3}{4} * n^2$
S_1 *total*	$3 * n^2$	$\frac{3}{4} * n^2$
S_2 *total*	$2 * n^2$	$\frac{1}{2} * n^2$
$S_1 + S_2$	$5 * n^2$	$\frac{5}{4} * n^2$

exhibits both poor temporal and poor spatial reuse. The problem is not the fault of the programmer; instead, it is inherent in how the computation can be expressed in Fortran 90. Fusing the K loops results in temporal locality for array B. In addition, the compiler is now able to apply loop interchange, significantly improving spatial locality for all the arrays. This transformation is illustrated in Figure 3(c).

4.3.1 Profitability of Loop Fusion

Loop fusion may improve reuse directly by moving accesses to the same cache line to the same loop iteration. Algorithm *RefGroup* discovers this reuse between two nests by treating the statements as if they already were in the same loop body. The two loop headers are *compatible* if the loops have the same number of iterations. Two nests are compatible at level l if the loops at level 1 to l are compatible and the headers are perfectly nested up to level l. To determine the profitability of fusing two compatible nests, we use the cost model as follows:

- Compute *RefGroup* and *LoopCost* as if all the statements were in the same nest, *i.e.,* fused.
- Compute *RefGroup* and *LoopCost* independently for each candidate and add the results.
- Compare the total *LoopCost*s.

If the fused *LoopCost* is lower, fusion alone will result in additional locality. As an example, fusing the two K loops in Figure 3 lowers the *LoopCost* for K from $5n^2$ to $3n^2$. Candidate loops for fusion need not be nested within a common loop. Note that the memory order for the fused loops may differ from the individual nests.

4.3.2 Loop Fusion to enable Loop Permutation

Loop fusion may also indirectly improve reuse in imperfect loop nests by providing a perfect nest that enables a loop permutation with better data locality. For instance, fusing the K loops in Figure 3 enables permuting the loop nest, improving spatial and temporal locality. Using the cost model, we detect that this transformation is desirable since *LoopCost* of the I loop is lower than the K loops, but memory order cannot be achieved because of the loop structure. We then test if fusion of all inner nests is legal and creates a perfect nest in which memory order can be achieved.

4.3.3 Loop Fusion Algorithm

Fusion thus serves two purposes:

1. to improve temporal locality, and
2. to fuse all inner loops, creating a nest that is permutable.

Previous research has shown that optimizing temporal locality for an adjacent set of n loops with compatible headers is NP-hard [KM93]; here all the headers are not necessarily compatible. We therefore apply a greedy strategy based on the depth of compatibility. We build a DAG from the candidate loops. The edges are dependences between the loops; the weight of an edge is the difference between the *LoopCost*s of the fused and unfused versions. We partition the nests into sets of compatible nests at the deepest levels possible. To yield the most locality, we first fuse nests with the deepest compatibility and temporal locality. Nests are fused only if is legal, *i.e.,* no dependences are violated between the loops or in the DAG. We update the graph, then fuse at the next level until all compatible sets are considered. This algorithm appears in Figure 4. The complexity of this algorithm is $O(m^2)$ time and space, where m is the number of candidate nests for fusion.

Figure 4: Fusion Algorithm

Fuse(L)
INPUT: $\mathcal{L} = l_1, \ldots, l_k$, nests that are fusion candidates

ALGORITHM:
  Build $\mathcal{H} = \{H_1, \ldots, H_j\}$, $H_i = \{h_k\}$ a set of compatible nests with *depth*$(H_i) \geq$ *depth*(H_{i+1})
  Build DAG $\mathcal{G}$ with dependence edges and weights
  for each $H_i = \{h_1 \ldots h_m\}$, i = 1 to j
    for $l_1 = h_1$ to h_m
      for $l_2 = h_2$ to l_1
        if (($\exists$ locality between l_1 and l_2)
          /* $\exists$ edge (l_1, l_2) with weight > 0 */
          & (it is legal to fuse them))
            fuse l_1 and l_2 and update $\mathcal{G}$
      endfor
    endfor
  endfor

4.3.4 Example: Erlebacher

The original hand-coded version of Erlebacher, a program solving PDEs using ADI integration with 3D arrays, mostly consists of single statement loops in memory order. We permuted the remaining loops into memory order, producing a distributed program version. Since the loops are fully distributed in this version, it resembles the output of a Fortran 90 scalarizer. We then applied *Fuse* to obtain more temporal locality. In Table 1, we measure the performance of the original program (Hand), the transformed program without fusion (Distributed), and the fused version (Fused).

Table 1: Performance of Erlebacher (in seconds)

Processor	Hand Coded	Memory Order Distributed	Memory Order Fused
Sun Sparc2	.806	.813	.672
Intel i860	.547	.548	.518
IBM RS/6000	.390	.400	.383

Fusion is always an improvement (of up to 17%) over the hand-coded and distributed versions. Since each statement is in a separate loop, many variables are shared between loops. Permuting the loops into memory order increases locality in each nest, but slightly degrades locality *between* nests, hence the degradation in performance of the distributed version compared to the original. Even though the benefits of fusion are additive rather than multiplicative as in loop permutation, its impact can be significant. In addition, its impact will increase as more programs are written with Fortran 90 array syntax.

4.4 Loop Distribution

Loop distribution separates independent statements in a single loop into multiple loops with identical headers. To maintain the meaning of the original loop, statements in a recurrence (a cycle in the dependence graph) must be placed in the same loop. Groups of statements that must be in the same loop are called *partitions*. In our system we only use loop distribution to indirectly improve reuse by enabling loop permutation on a nest that is not permutable[3]. Statements in different partitions may prefer different memory orders that are achievable after distribution. The algorithm *Distribute* appears in Figure 5. It divides the statements into the finest granularity partitions and tests if that enables loop permutation. It performs the smallest amount of distribution that still enables permutation. For a nest of depth m, it starts with the loop at level $m - 1$ and works out to the outermost loop, stopping if successful.

[3]Distribution could also be effective if there is no temporal locality between partitions and the accessed arrays are too numerous to fit in cache at once, or register pressure is a concern. We do not address these issues here.

Figure 5: Distribution Algorithm

Distribute($\mathcal{L}$, $\mathcal{S}$)
INPUT: $\mathcal{L} = \{l_1, \ldots, l_m\}$, a loop nest containing $\mathcal{S} = \{s_1, \ldots, s_k\}$ statements

ALGORITHM:
  for $j = m - 1$ to 1
    Restrict the dependence graph to δ carried at level j or deeper and loop independent δ
    Divide $\mathcal{S}$ into finest partitions $\mathcal{P} = \{p_1, \ldots, p_m\}$ *s.t.* if $s_r, s_t \in$ a recurrence, $s_r, s_t \in p_i$.
    compute *MemoryOrder*$_i$ for each p_i
    if ($\exists\, i \mid$ *MemoryOrder*$_i$ is achievable with distribution and permutation)
      perform distribution and permutation
      return
  endfor

We only call *Distribute* if memory order cannot be achieved on a nest and not all the inner nests can be fused (see Section 4.5). *Distribute* tests if distribution will enable memory order to be achieved for any of the partitions. The dependence structure required to test for loop permutation is created by restricting its test to dependences on statements in the partition of interest. We thus perform distribution only if it combines with permutation to improve the actual *LoopCost*. The algorithm's complexity is dominated by the time to determine the *LoopCost* of the individual partitions. See Section 4.5.1 for an example.

4.5 Compound Transformation Algorithm

The driving force behind our application of compound loop transformations is to minimize actual *LoopCost* by achieving memory order for as many statements in the nest as possible. The algorithm *Compound* uses permutation, fusion, distribution, and reversal as needed to place the loop that provides the most reuse at the innermost position for each statement.

Algorithm *Compound* in Figure 6 considers adjacent loop nests. It first optimizes each nest independently, then applies fusion between the resulting nests when legal and temporal locality is improved. To optimize a nest, the algorithm begins by computing memory order and determining if the loop containing the most reuse can be placed innermost. If it can, the algorithm does so and goes on to the next loop. Otherwise, it tries to enable permutation into memory order by fusing all inner loops to form a perfect nest. If fusion cannot enable memory order, the algorithm tries distribution. If distribution succeeds in enabling memory order, several new nests may be formed. Since the distribution algorithm divides the statements into the finest partitions, these nests are candidates for fusion to recover temporal locality.

Complexity. Ignoring distribution for a moment, the complexity of the compound algorithm is $O(n) + O(n^2) + O(m^2)$ time, where n is the maximum number of loops in a nest, and m is the maximum number of adjacent nests in the original program. $O(n)$ non-trivial steps are needed to evaluate the locality of the statements in each nest. $O(n^2)$ simple steps result in the worst case when finding a

Figure 6: Compound Loop Transformation Algorithm

Compound($\mathcal{N}$)
INPUT: $\mathcal{N} = \{n_1, \ldots n_k\}$, adjacent loop nests
ALGORITHM:
  for i = 1 to k
    Compute MemoryOrder (n_i)
    if (*Permute*(n_i) places inner loop in memory order)
      continue
    else if (n_i is not a perfect nest & contains only
      adjacent loops m_j)
        if (*FuseAll*(m_j,l) and *Permute*(l)
          places inner loop in memory order)
            continue
    else if (*Distribute*(n_i,l))
      Fuse(l)
  end for
  Fuse($\mathcal{N}$)

legal permutation and $O(m^2)$ steps result from building the fusion problem. However, because distribution produces more adjacent nests that are candidates for fusion, m includes the additional adjacent nests created by distribution. In practice, this increase was negligible; a single application of distribution never created more than 3 new nests.

4.5.1 Example: Cholesky Factorization

Consider optimizing the Cholesky Factorization kernel in Figure 7(a) with algorithm *Compound*. *LoopCost* determines that memory order for the nest is KJI, ranking the nests from lowest cost to highest (KJI, JKI, KIJ, IKJ, JIK, IJK). Because KJI cannot be achieved with permutation alone and fusion is of no help here, *Compound* calls *Distribute*. Since the loop is of depth 3, *Distribute* starts by testing distribution at depth 2, the I loop. S_2 and S_3 go into separate partitions (there is no recurrence between them at level 2 or deeper). Memory order of S_3 is also KJI. Distribution of the I loop places S_3 alone in a IJ nest where I and J may be legally interchanged into memory order, as shown in Figure 7(b). Note that our system handles the permutation of both triangular and rectangular nests.

To gather performance results for Cholesky, we generated all possible loop permutations; they are all legal. For each permutation, we applied the minimal amount of loop distribution necessary. (Wolfe enumerates these loop organizations [Wol91].) Compared to matrix multiply, there are more variations in observed and predicted behavior. These variations are due to the triangular loop structure; however, *Compound* still attains the loop structure with the best performance.

Figure 7: Cholesky Factorization

(a) {KIJ form}

```
        DO K=1,N
S1        A(K,K) = SQRT(A(K,K))
          DO I=K+1,N
S2          A(I,K) = A(I,K)/A(K,K)
            DO J=K+1,I
S3            A(I,J) = A(I,J)-A(I,K)*A(J,K)
```

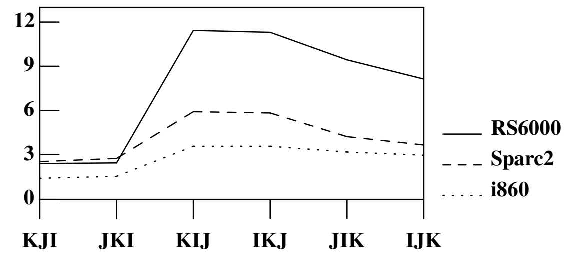
(b) $\Downarrow$ {KJI form} *Loop Distribution & Triangular Interchange* $\Downarrow$

```
        DO K=1,N
          A(K,K) = SQRT(A(K,K))
          DO I=K+1,N
            A(I,K)=A(I,K)/A(K,K)
          DO J=K,N
            DO I=J+1,N
              A(I,J+1) = A(I,J+1)-A(I,K)*A(J+1,K)
```

LoopCost

Refs	K	J	I
A(K,K)	$n * n$	—	$1 * n$
A(I,K)	$n * n^2$	$1 * n^2$	$\frac{1}{4}n * n^2$
A(I,J)	$1 * n^2$	$n * n^2$	$\frac{1}{4}n * n^2$
A(J,K)	$n * n^2$	$\frac{1}{4}n * n^2$	$1 * n^2$
total	$2n^3 + 2n^2$	$\frac{5}{4}n^3 + n^2$	$\frac{1}{2}n^3 + n^2 + n$
S_2 *total*	$2n^2$	—	$\frac{1}{4}n^2 + n$
S_3 *total*	$2n^3 + n^2$	$\frac{5}{4}n^3 + n^2$	$\frac{1}{2}n^3 + n^2$

Execution times (in seconds)
300 x 300



5 Experimental Results

To validate our optimization strategy, we implemented our algorithms, executed the original and transformed program versions on our test suite and simulated cache hit rates. To measure our ability to improve locality, we determined the best locality possible if correctness could be ignored. We collected statistics on the data locality in the original, transformed, and *ideal* programs.

5.1 Methodology

We implemented the cost model, the transformations, and the algorithms described above in Memoria, the Memory Compiler in the ParaScope Programming Environment [Car92, CHH+93, CK94, KMT93]. Memoria is a source-to-source translator that analyzes Fortran programs and transforms them to improve their cache performance. To increase the precision of dependence analysis, the compiler performs auxiliary induction variable substitution, constant propagation, forward expression propagation, and dead code elimination. It also determines if scalar expansion will further enable distribution. Since scalar expansion is not integrated in the current version of the transformer, we applied it by hand when directed by the compiler. Memoria then used the resulting code and dependence graph to gather statistics and perform data locality optimizations using the algorithm *Compound*.

For our test suite, we used 35 programs from the Perfect Benchmarks, the SPEC benchmarks, the NAS kernels, and some miscellaneous programs. They ranged in size from 195 to 7608 non-comment lines. Their execution times on the IBM RS/6000 ranged from seconds to a couple of hours.

5.2 Transformation Results

In Table 2, we report the results of transforming the loop nests of each program. Table 2 first lists the number of non-comment lines (Lines), the number of loops (Loops), and the number of loop nests (Nests) for each program. Only loop nests of depth 2 or more are considered for transformation. **MemoryOrder** and **Inner Loop** entries reflect the percentage of loop nests and inner loops,

Table 2: Memory Order Statistics

Program	Lines	Loops	Nests	MemoryOrder Orig	MemoryOrder Perm	MemoryOrder Fail	Inner Loop Orig	Inner Loop Perm	Inner Loop Fail	Loop Fusion C	Loop Fusion A	Loop Distribution D	Loop Distribution R	LoopCost Ratio Final	LoopCost Ratio Ideal
				%	percentages	%									
Perfect Benchmarks															
adm	6105	219	106	52	16	32	53	16	31	0	0	1	2	2.54	6.10
arc2d	3965	152	75	55	28	17	65	34	1	35	12	1	2	2.21	4.14
bdna	3980	104	56	75	18	7	75	18	7	4	2	3	6	2.31	2.51
dyfesm	7608	164	80	63	15	22	65	19	16	2	1	0	0	3.08	8.62
flo52	1986	149	76	83	17	0	95	5	0	4	1	0	0	1.72	1.79
mdg	1238	25	12	83	8	8	83	8	8	0	0	0	0	1.11	1.70
mg3d	2812	88	40	95	3	3	98	0	2	0	0	1	2	1.00	1.13
ocean	4343	115	56	82	13	5	84	13	4	2	1	3	6	2.05	2.20
qcd	2327	94	45	53	11	36	58	16	15	0	0	0	0	4.98	6.10
spec77	3885	255	162	64	7	29	66	7	27	0	0	0	0	2.32	5.58
track	3735	57	32	50	16	34	56	19	25	2	1	1	2	1.99	7.95
trfd	485	67	29	52	0	48	66	0	34	0	0	0	0	1.00	14.81
SPEC Benchmarks															
dnasa7	1105	111	50	64	14	22	74	16	10	5	2	1	2	2.08	2.95
doduc	5334	60	33	6	6	88	6	6	88	0	0	4	12	1.89	14.25
fpppp	2718	23	8	88	12	0	88	12	0	0	0	0	0	1.03	1.03
hydro2d	4461	110	55	100	0	0	100	0	0	44	11	0	0	1.00	1.00
matrix300	439	4	2	50	50	0	50	50	0	0	0	1	2	4.50	4.50
mdljdp2	4316	4	1	0	0	100	0	0	100	0	0	0	0	1.00	1.05
mdljsp2	3885	4	1	0	0	100	0	0	100	0	0	0	0	1.00	1.02
ora	453	6	3	100	0	0	100	0	0	0	0	0	0	1.00	1.00
su2cor	2514	84	36	42	19	39	42	19	39	0	0	4	8	3.51	5.30
swm256	487	16	8	88	12	0	88	12	0	0	0	0	0	4.91	4.91
tomcatv	195	12	6	100	0	0	100	0	0	7	2	0	0	1.00	1.00
NAS Benchmarks															
appbt	4457	181	87	98	0	2	100	0	0	3	1	0	0	1.00	1.26
applu	3285	155	71	73	3	24	79	6	15	3	1	2	6	1.35	8.03
appsp	3516	184	84	73	12	15	80	12	8	8	4	0	0	1.25	4.34
buk	305	0	0	0	0	0	0	0	0	0	0	0	0	1.00	1.00
cgm	855	11	6	0	0	100	0	0	100	0	0	0	0	1.00	2.75
embar	265	3	2	50	0	50	50	0	50	0	0	0	0	1.00	1.12
fftpde	773	40	18	89	0	11	100	0	0	0	0	0	0	1.00	1.00
mgrid	676	43	19	89	11	0	100	0	0	3	1	1	2	1.00	1.00
Miscellaneous Programs															
erlebacher	870	75	30	83	13	4	100	0	0	28	11	0	0	1.00	1.00
linpackd	797	8	4	75	0	25	75	0	25	3	1	0	0	1.00	1.10
simple	1892	39	22	86	9	5	86	9	5	6	2	0	0	2.48	2.72
wave	7519	180	85	58	29	13	65	29	6	70	26	0	0	4.26	4.30
totals	—	2644	1400	69	11	20	74	11	15	229	80	23	52	—	—

respectively, that are:

Orig: originally in memory order,
Perm: permuted into memory order, or
Fail: fail to achieve memory order.

The percentage of loop nests in the program that are in memory order after transformation is the sum of the original and the permuted entries. Similarly for the inner loop, the sum of the original and the permuted entries is the percent of nests where the most desirable innermost loop is positioned correctly.

Table 2 also lists the number of times that fusion and distribution were applied by the compound algorithm. Either fusion, distribution, or both were applied to 22 out of the 35 programs.

In the **Loop Fusion** column,

C is the number of candidate nests for fusion, and
A is the number of nests that were actually fused.

Candidate nests for fusion were adjacent nests, where at least one pair of nests were compatible. Fusion improved group-temporal locality for these programs; it did not find any opportunities to enable interchange. There were 229 adjacent loop nests that were candidates for fusion and of these, 80 were fused with one or more other nests to improve reuse. Fusion was applicable in 17 programs and completely fused nests of depth 2 and 3. In *Wave* and *Arc2d*, *Compound* fused 26 and 12 nests respectively.

In the **Loop Distribution** column,

D is the number of loop nests distributed to achieve a better loop permutation, and
R is the number of nests that resulted.

The *Compound* algorithm only applied distribution when it enabled permutation to attain memory order in a nest or in the innermost loop for at least one of the resultant nests. *Compound* applied distribution in 12 of the 35 programs. On 23 nests, distribution enabled loop permutation to position the inner loop or the entire nest correctly, creating 29 additional nests. In *Bdna*, *Ocean*, *Applu* and *Su2cor* 6 or more nests resulted.

LoopCost Ratio in Table 2 estimates the potential reduction in LoopCost for the final transformed program (**Final**) and the *ideal* program (**Ideal**). The ideal program achieves memory order for every nest without regard to dependence constraints or limitations in the implementation. By ignoring correctness, it is in some sense the best data locality one could achieve. For the final and ideal versions, the average ratio of original LoopCost to transformed LoopCost is listed. These values reveal the potential for locality improvement.

Memoria may not obtain memory order due to the following: (1) loop permutation is illegal due to dependences; (2) loop distribution followed by permutation is illegal due to dependences; (3) the loop bounds are too complex, *i.e.,* not rectangular or triangular. For the 20% of nests where the compiler could not achieve memory order, 87% were because permutation and then distribution followed by permutation could not be applied because of dependence constraints. The rest were because the loop bounds were too complex. More sophisticated dependence testing techniques may enable the algorithms to transform a few more nests.

5.3 Coding Styles

Imprecise dependence analysis is a factor in limiting the potential for improvements in our application suite. For example, dependence analysis for the program *Cgm* cannot expose potential data locality for our algorithm because of imprecision due to the use of index arrays. The program *Mg3d* is written with linearized arrays. This coding style introduces symbolics into the subscript expressions and again makes dependence analysis imprecise. The inability to analyze the use of index arrays and linearized arrays prevents many optimizations and is not a deficiency specific to our system.

Other coding styles may also inhibit optimization in our system. For example, *Linpackd* and *Matrix300* are written in a modular style with singly nested loops enclosing function calls to routines which also contain singly nested loops. To improve programs written in this style requires interprocedural optimization [CHK93, HKM91]; these optimizations are not currently implemented in our translator.

Many loop nests (69%) in the original programs are already in memory order, and even more (74%) have the loop carrying the most reuse in the innermost position. This result indicates that scientific programmers often pay attention to data locality; however, there are many opportunities for improvement. Our compiler was able to permute an additional 11% of the loop nests into memory order, resulting in a total 80% of the nests in memory order and a total of 85% of the inner loops in memory order position. Memoria improved data locality for one or more nests in 66% of the programs.

5.4 Successful Transformation

We illustrate our ability to transform for data locality by program in Figures 8 and 9. The figures characterize the programs by the percentage of their nests and inner loops that are originally in memory order and that are transformed into memory order. In Figure 8, half of the original programs have fewer than 70% of their nests in memory order. In the transformed version, 29% have fewer than 70% of their nests in memory order. Over half now have 80% or more of their nests in memory order. The results in Figure 9 are more dramatic. The majority of programs can be transformed such that 90% or more of their inner loops are positioned correctly for the best locality. Our transformation algorithms determine and achieve memory order in the majority of nests and programs.

Unfortunately, our ability to successfully transform programs may not result in run-time improvements for several reasons: data sets for benchmark programs tend to be small enough to fit in cache, the transformed loop nests may be cpu-bound instead of memory-bound, and the optimized portions of the program may not significantly contribute to the overall execution time.

Figure 8: Achieving Memory Order for Loop Nests

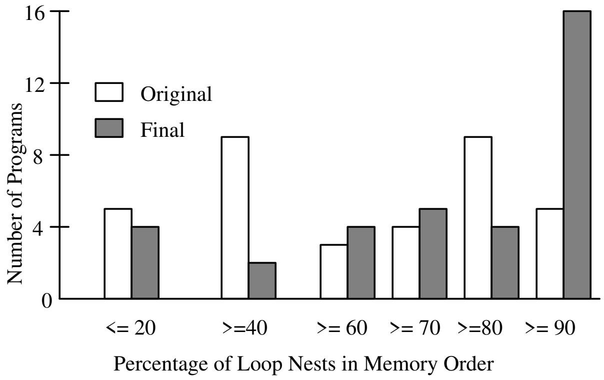
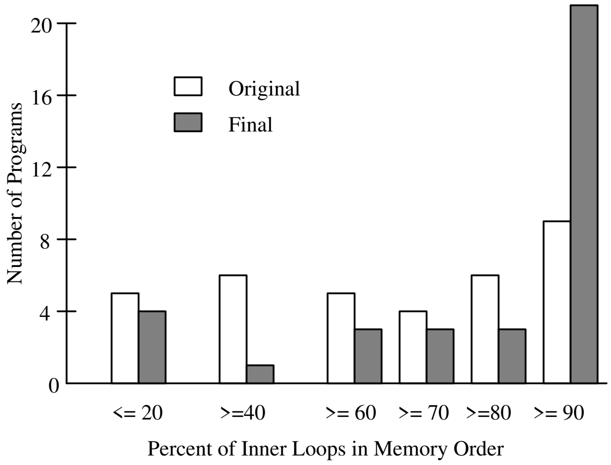


Figure 9: Achieving Memory Order for the Inner Loop



5.5 Performance Results

In Table 3, we present the performance of our test suite running on an IBM RS/6000 model 540 with a 64KB cache, 4-way set associative replacement policy, and 128 byte cache lines. We used the standard IBM RS/6000 Fortran 77 compiler with the -O option to compile both the original program and the version produced by our automatic source-to-source transformer. All applications successfully compiled and executed on the RS/6000. For those applications not listed in Table 3, no performance improvement or degradation occurred.

Table 3 shows a number of applications with significant performance improvements (*Arc2d*, *Dnasa7*, and *Simple*). These results indicate that data locality optimizations are particularly effective for scalarized vector programs, since these programs are structured to emphasize vector operations rather than cache line reuse. However, the predicted improvements did not materialize for many of the programs. To explore these results, we simulated cache behavior to determine cache hit rates for out test suite.

We simulated *cache1*, an RS/6000 cache (64KB, 4-way set associative, 128 byte cache lines), and *cache2*, an i860 cache (8KB, 2-way set associative, 32 byte cache lines). We determined the change in the hit rates both for just the optimized procedures and for the entire program. The resulting measured rates are presented in Table 4. Places where the compiler affected cache hit rates by $\geq .1\%$ are emboldened for greater emphasis. For the *Final* columns we chose the better of the fused and unfused versions for each program.

Table 3: Performance Results (in seconds)

RS/6000 with 64KB, 4-way set associative cache and cache line size of 128 byte.

Program	Original	Transformed	Speedup
Perfect Benchmarks			
arc2d	410.13	190.69	2.15
dyfesm	25.42	25.37	1.00
flo52	62.06	61.62	1.01
SPEC Benchmarks			
dnasa7 (btrix)	36.18	30.27	1.20
dnasa7 (emit)	16.46	16.39	1.00
dnasa7 (gmtry)	155.30	17.89	8.68
dnasa7 (vpenta)	149.68	115.62	1.29
NAS Benchmarks			
applu	146.61	149.49	0.98
appsp	361.43	337.84	1.07
Misc Programs			
simple	963.20	850.18	1.13
linpackd	159.04	157.48	1.01
wave	445.94	414.60	1.08

As illustrated in Table 4, the reason more programs did not improve on the RS/6000 is due to high hit ratios in the original programs caused by small data set sizes. When the cache is reduced to 8K, the optimized portions have more significant improvements. For instance, whole program hit rates for *Dnasa7* and *Appsp* show significant improvements after optimization for the smaller cache even though they barely changed in the larger cache. Our optimizations obtained improvements in whole program hit rates for *Adm, Arc2d, Dnasa7, Hydro2d, Appsp, Erlebacher, Simple*, and *Wave*. Improvements in the optimized loop nests were more dramatic, but did not always carry over to the entire program because of the presence of unoptimized loops.

We measured hit ratios both with and without applying loop fusion. For the 8K cache, fusion improved whole program hit rates for *Hydro2d*, *Appsp*, and *Erlebacher* by 0.51%, 0.24%, and 0.95%, respectively. We were surprised to improve *Linpackd*'s performance with fusion by 5.3% on the subroutine *matgen* and by 0.02% for the entire program. *Matgen* is an initialization routine whose performance is not usually measured. Unfortunately, fusion also lowered hit rates *Track*, *Dnasa7*, and *Wave*; the degradation may be due to added cache conflict and capacity misses after loop fusion. To recognize and avoid these situations requires cache capacity and interference analysis similar to that performed for evaluating loop tiling [LRW91]. Because our fusion algorithm only attempts to optimize reuse at the innermost loop level, it may sometimes merge array references that interfere or overflow cache. We intend to correct this deficiency in the future.

Our results are very favorable when compared to Wolf's results, though direct comparisons are difficult because he combines tiling with cache optimizations and reports improvements only relative to programs with scalar replacement [Wol92]. Wolf applied permutation, skewing, reversal and tiling to the Perfect Benchmarks and *Dnasa7* on a DECstation 5000 with a 64KB direct-map cache. His results show performance degradations or no change in all but *Adm*, which showed a small (1%) improvement in execution time. Our transformations did not degrade performance on any of the Perfect programs and performance of *Arc2d* was significantly improved.

Our results on the routines in *Dnasa7* are similar to Wolf's, both showing improvements on *Btrix, Gmtry,* and *Vpenta*. Wolf improved *Mxm* by about 10% on the DECstation, but slightly degraded performance on the i860. Wolf slowed *Cholesky* by about 10%

Table 4: Simulated Cache Hit Rates

Cache1: 64K cache, 4-way, 128 byte cache line (RS/6000)
Cache2: 8K cache, 2-way, 32 byte cache line (i860)
Cold misses are not included

	Optimized Procedures				Whole Program			
	Cache 1		Cache 2		Cache 1		Cache 2	
Program	Orig	Final	Orig	Final	Orig	Final	Orig	Final
Perfect Benchmarks								
adm	100	100	**97.7**	**97.8**	99.95	99.95	**98.48**	**98.58**
arc2d	**89.0**	**98.5**	**68.3**	**91.9**	**95.30**	**98.66**	**88.58**	**93.61**
bdna	100	100	100	100	99.45	99.45	97.32	97.32
dyfesm	100	100	100	100	99.98	99.97	97.02	96.95
flo52	99.6	99.6	**96.7**	**96.3**	98.77	98.77	93.84	93.80
mdg	100	100	87.4	87.4	——	——	——	——
mg3d	**98.8**	**99.7**	**95.3**	**98.7**	——	——	——	——
ocean	100	100	**93.0**	**92.8**	99.36	99.36	93.71	93.72
qcd	100	100	100	100	99.83	99.83	98.85	98.79
spec77	100	100	100	100	99.28	99.28	93.79	93.78
track	100	100	100	100	99.81	99.81	97.49	97.54
trfd	99.9	99.9	93.7	93.7	99.92	99.92	96.43	96.40
SPEC Benchmarks								
dnasa7	**83.2**	**92.7**	**54.5**	**73.9**	99.26	99.27	**85.45**	**88.76**
doduc	100	100	95.5	95.5	99.77	99.77	95.92	95.92
fpppp	100	100	100	100	99.99	99.99	98.34	98.34
hydro2d	**97.9**	**98.3**	**90.2**	**91.9**	**98.36**	**98.48**	**92.77**	**93.28**
matrix300	99.7	99.7	**91.6**	**92.1**	93.26	93.26	81.66	81.67
su2cor	100	100	**99.2**	**99.8**	98.83	98.83	70.41	70.41
swm256	100	100	100	100	98.83	98.84	**81.00**	**81.11**
tomcatv	97.8	97.8	87.3	87.3	99.20	99.20	95.26	95.25
NAS Benchmarks								
applu	99.9	99.9	99.4	99.4	99.38	99.36	97.22	97.14
appsp	**90.5**	**92.9**	**88.5**	**89.0**	99.33	99.39	**96.04**	**96.43**
mgrid	**99.3**	**99.8**	**91.6**	**92.1**	99.65	99.65	96.04	96.04
Miscellaneous Programs								
erlebacher	**99.4**	**99.8**	**94.0**	**96.8**	**98.00**	**98.25**	**92.11**	**93.36**
linpackd	**98.7**	**100**	**94.7**	**100**	98.93	98.94	95.58	95.60
simple	**91.0**	**99.1**	**84.3**	**93.7**	**97.35**	**99.34**	**93.33**	**95.65**
wave	**98.2**	**99.9**	**82.9**	**95.9**	**99.74**	**99.82**	**87.31**	**88.09**

on the DECstation and by a slight amount on the i860. We neither improve or degrade either kernel. More direct comparisons are not possible because Wolf does not present cache hit rates and the execution times were measured on different architectures.

5.6 Data Access Properties

To further interpret our results, we measured the data access properties for our test suite. For the applications that we significantly improved on the RS/6000 (*Arc2d, Dnasa7, Appsp, Simple* and *Wave*), we present the data access properties in Table 5.[4] We report the locality statistics for the original, ideal memory order, and final versions of the programs. **Locality of Reference Group** classifies the percentage of RefGroups displaying each form of self reuse as invariant (Inv), unit-stride (Unit), or none (None). (Group) contains the percentage of RefGroups constructed partly or completely using group-spatial reuse. The amount of group reuse is indicated by measuring the average number of references in each RefGroup (Refs/Group), where a RefGroup size greater than 1 implies group-temporal reuse and occasionally group-spatial reuse. The amount of group reuse is presented for each type of self reuse and their average (Avg). The **LoopCost Ratio** column estimates the potential improvement as an average (Avg) over all the nests and a weighted

[4] The data access properties for all the programs are presented elsewhere [CMT94].

Table 5: Data Access Properties

Program		Locality of Reference Groups: % Groups Inv	Unit	None	Group	Refs/Group Inv	Unit	None	Avg	LoopCost Ratios Avg	Wt
arc2d	*original*	3	53	44	1	1.53	1.23	1.26	1.25		
	final	3	77	20	0	2.12	1.34	1.00	1.29	2.21	2.16
	ideal	14	66	20	0	1.72	1.31	1.00	1.30	4.14	4.73
dnasa7	*original*	5	48	47	0	1.41	1.48	1.16	1.33		
	final	8	57	35	0	1.33	1.48	1.10	1.34	2.08	2.27
	ideal	35	37	28	0	1.61	1.27	1.07	1.34	2.95	3.33
appsp	*original*	0	38	62	0	0	1.04	1.08	1.06		
	final	0	49	51	0	0	1.03	1.09	1.06	1.25	1.24
	ideal	8	44	48	0	1.49	1.03	1.02	1.06	4.34	4.43
simple	*original*	0	93	7	0	0	2.25	1.85	2.22		
	final	0	98	2	0	0	2.26	1.00	2.23	2.48	2.48
	ideal	1	97	2	0	1.50	2.27	1.00	2.23	2.72	2.72
wave	*original*	6	47	47	1	1.95	1.48	1.27	1.41		
	final	1	71	28	0	2.00	1.55	1.02	1.41	4.26	4.25
	ideal	3	70	27	0	1.63	1.55	1.01	1.41	4.30	4.28
all programs	*original*	3	37	60	0	1.53	1.26	1.15	1.23		
	final	3	44	53	0	1.52	1.27	1.05	1.23	—	—
	ideal	8	41	51	0	1.23	1.26	1.03	1.23	—	—

average (Wt) uses nesting depth. The last row contains the totals for all the programs.

Table 5 also reveals that each of these applications had a significant gain in self-spatial reuse (Unit) over the original program. Because of the relatively long cache lines on the RS/6000, spatial locality was the key to getting good cache performance. Although programmers can make the effort to ensure unit-stride access on RS/6000 applications, we have shown that our optimization strategy makes this unnecessary. By having the compiler compute the machine-dependent loop ordering, a variety of coding styles can be run efficiently without additional programmer effort.

The **all programs** row in Table 5 reveals that on average fewer than two references exhibited group-temporal reuse in the inner loop, and no references displayed group-spatial reuse. Instead, most programs exhibit self-spatial reuse. The ideal program exhibits significantly more invariant reuse than the original or final. Invariant reuse typically occurs on loops with reductions and time-step loops that are often involved in recurrences and cannot be permuted. However, tiling may be able to exploit some of the invariant reuse carried by outer loops.

5.7 Analysis of Individual Programs

Below, we examine *Arc2d, Simple, Gmtry* (three of the applications that we improved) and *Applu* (the only application with a degradation in performance. We note specific coding styles that our system effectively ported to the RS/6000.

Arc2d is a fluid-flow solver from the Perfect benchmarks. The main computational routines exhibit poor cache performance due to non-unit stride accesses. The main computational loop is an imperfect loop nest with four inner loops, two with nesting depth 2 and two with nesting depth 3. Our algorithm is able to achieve a factor of 6 improvement on the main loop nest by attaining unit-stride accesses to memory in the two loops with nesting depth 3. This improvement alone accounted for a factor of 1.9 on the whole application. The additional improvement illustrated in Table 3 is attained similarly by improving less time-critical routines. Our optimization strategy obviated the need for the programmer to select the "correct" loop order for performance.

Simple is a two-dimensional hydrodynamics code. It contains two loops that are written in a "vectorizable" form (*i.e.,* a recurrence is carried by the outer loop rather than the innermost loop). These loops exhibited poor cache performance. *Compound* reorders these loops for data locality (both spatial and temporal) rather than vectorization to achieve the improvements shown in Table 3. In this case, the improvements in cache performance far outweigh the potential loss in low-level parallelism when the recurrence is carried by the innermost loop. To regain any lost parallelism, unroll-and-jam can be applied to the outermost loop [CCK88, Car92]. Finally, it is important to note that the programmer was allowed to write the code in a form for one type of machine and still attain machine-independent performance through the use of compiler optimization.

Gmtry, a SPEC benchmark kernel from *Dnasa7*, performs Gaussian elimination across rows, resulting in no spatial locality. Although this may have been how the author viewed Gaussian elimination conceptually, it translated to poor performance. Distribution and permutation are able to achieve unit-stride accesses in the innermost loop. The programmer is therefore allowed to write the code in a form that she or he understands, while the compiler handles the machine-dependent performance details.

Applu suffers from a tiny degradation in performance (1%). The two leading dimensions of the main data arrays are very small (5×5). While our model predicts better performance for unit-stride access to the arrays, the small array dimensions give the original reductions better performance on the RS/6000. Locality within the two innermost loops is not a problem.

6 Tiling

Permuting loops into memory order maximizes estimated short-term cache-line reuse across iterations of inner loops. Assuming that the cache size is relatively large, the compiler can apply *loop tiling*, a combination of strip-mining and loop interchange, to capture long-term reuse at outer loops [IT88, LRW91, WL91, Wol87]. Tiling must be applied judiciously because it affects scalar optimizations, increases loop overhead, and may decrease spatial reuse at tile boundaries. Our cost model provides us with the key insight to guide tiling—the primary criterion for tiling is to create loop-invariant references with respect to the target loop. These references access significantly fewer cache lines than both consecutive and non-consecutive references, making tiling worthwhile despite the potential loss of spatial reuse at tile boundaries. For machines with long cache lines, it may also be advantageous to tile outer loops if they carry many unit-stride references, such as when transposing a matrix. In the future, we intend to study the cumulative effects of optimizations presented in this paper with tiling, unroll-and-jam, and scalar replacement.

7 Conclusion

This paper presents a comprehensive approach to improving data locality and is the first to combine loop permutation, fusion, distribution, and reversal into an integrated algorithm. Because we accept some imprecision in the cost model, our algorithms are simple and inexpensive in practice, making them ideal for use in a compiler. More importantly, the imprecision in our model is not a factor in the compiler's ability to exploit data locality. The empirical results presented in this paper validate the accuracy of our cost model and algorithms for selecting the best loop structure for data locality. In addition, they show this approach has wide applicability for existing Fortran programs regardless of their original target architecture, but particularly for vector and Fortran 90 programs. We believe this is a significant step towards achieving good performance with machine-independent programming.

Acknowledgments

We wish to thank Ken Kennedy for providing the impetus and guidance for much of this research. We are grateful to the ParaScope research group at Rice University for the software infrastructure on which this work depends. In particular, we appreciate the assistance of Nathaniel McIntosh on simulations. We acknowledge the Center for Research on Parallel Computation at Rice University for supplying most of the computing resources for our experiments and simulations.

References

[AS79] W. Abu-Sufah. *Improving the Performance of Virtual Memory Computers*. PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1979.

[Car92] S. Carr. *Memory-Hierarchy Management*. PhD thesis, Dept. of Computer Science, Rice University, September 1992.

[CCK88] D. Callahan, J. Cocke, and K. Kennedy. Estimating interlock and improving balance for pipelined machines. *Journal of Parallel and Distributed Computing*, 5(4):334–358, August 1988.

[CCK90] D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. In *Proceedings of the SIGPLAN '90 Conference on Program Language Design and Implementation*, White Plains, NY, June 1990.

[CHH$^+$93] K. Cooper, M. W. Hall, R. T. Hood, K. Kennedy, K. S. McKinley, J. M. Mellor-Crummey, L. Torczon, and S. K. Warren. The ParaScope parallel programming environment. *Proceedings of the IEEE*, 81(2):244–263, February 1993.

[CHK93] K. Cooper, M. W. Hall, and K. Kennedy. A methodology for procedure cloning. *Computer Languages*, 19(2):105–117, February 1993.

[CK94] S. Carr and K. Kennedy. Scalar replacement in the presence of conditional control flow. *Software—Practice and Experience*, 24(1):51–77, January 1994.

[CMT94] S. Carr, K. S. McKinley, and C.-W. Tseng. Compiler optimizations for improving data locality. Technical Report TR94-234, Dept. of Computer Science, Rice University, July 1994.

[FST91] J. Ferrante, V. Sarkar, and W. Thrash. On estimating and enhancing cache effectiveness. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing, Fourth International Workshop*, Santa Clara, CA, August 1991. Springer-Verlag.

[GJG88] 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, October 1988.

[GKT91] G. Goff, K. Kennedy, and C.-W. Tseng. Practical dependence testing. In *Proceedings of the SIGPLAN '91 Conference on Program Language Design and Implementation*, Toronto, Canada, June 1991.

[HKM91] M. W. Hall, K. Kennedy, and K. S. McKinley. Interprocedural transformations for parallel code generation. In *Proceedings of Supercomputing '91*, Albuquerque, NM, November 1991.

[IT88] F. Irigoin and R. Triolet. Supernode partitioning. In *Proceedings of the Fifteenth Annual ACM Symposium on the Principles of Programming Languages*, San Diego, CA, January 1988.

[KKP$^+$81] D. Kuck, R. Kuhn, D. Padua, B. Leasure, and M. J. Wolfe. Dependence graphs and compiler optimizations. In *Conference Record of the Eighth Annual ACM Symposium on the Principles of Programming Languages*, Williamsburg, VA, January 1981.

[KM92] K. Kennedy and K. S. McKinley. Optimizing for parallelism and data locality. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, Washington, DC, July 1992.

[KM93] K. Kennedy and K. S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

[KMT93] K. Kennedy, K. S. McKinley, and C.-W. Tseng. Analysis and transformation in an interactive parallel programming tool. *Concurrency: Practice & Experience*, 5(7):575–602, October 1993.

[LP92] W. Li and K. Pingali. Access normalization: Loop restructuring for NUMA compilers. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, MA, October 1992.

[LRW91] 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*, Santa Clara, CA, April 1991.

[McK92] K. S. McKinley. *Automatic and Interactive Parallelization*. PhD thesis, Dept. of Computer Science, Rice University, April 1992.

[War84] J. Warren. A hierachical basis for reordering transformations. In *Conference Record of the Eleventh Annual ACM Symposium on the Principles of Programming Languages*, Salt Lake City, UT, January 1984.

[WL91] M. E. Wolf and M. Lam. A data locality optimizing algorithm. In *Proceedings of the SIGPLAN '91 Conference on Program Language Design and Implementation*, Toronto, Canada, June 1991.

[Wol87] M. J. Wolfe. Iteration space tiling for memory hierarchies, December 1987. Extended version of a paper which appeared in *Proceedings of the Third SIAM Conference on Parallel Processing*.

[Wol91] M. J. Wolfe. The Tiny loop restructuring research tool. In *Proceedings of the 1991 International Conference on Parallel Processing*, St. Charles, IL, August 1991.

[Wol92] M. E. Wolf. *Improving Locality and Parallelism in Nested Loops*. PhD thesis, Dept. of Computer Science, Stanford University, August 1992.