Improving Data Locality with Loop Transformations

KATHRYN S. McKINLEY
University of Massachusetts at Amherst
and
STEVE CARR
Michigan Technological University
and
CHAU-WEN TSENG
University of Maryland at College Park

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 article, 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 illustrate that for kernels our model and algorithm can select and achieve the best loop structure for a nest. For over 30 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.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*compilers*; *optimization*

General Terms: Languages, Performance

Additional Key Words and Phrases: Cache, compiler optimization, data locality, loop distribution, loop fusion, loop permutation, loop reversal, loop transformations, microprocessors, simulation

---

Steve Carr was supported by NSF grant CCR-9409341 and Hewlett-Packard Company. Chau-Wen Tseng was supported in part by an NSF CISE Postdoctoral Fellowship in Experimental Science. The authors initiated this research at Rice University.

Authors' addresses: K. S. McKinley, Computer Science Department, LGRC, University of Massachusetts, Amherst, MA 01003-4610; email: mckinley@cs.umass.edu; S. Carr, Department of Computer Science, Michigan Technological University, Houghton, MI 49931-1295; email: carr@cs.mtu.edu; C.-W. Tseng, Department of Computer Science, University of Maryland, College Park, MD 20742; email: tseng@cs.umd.edu.

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.

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 [Irigoin and Triolet 1988]. Knowledge of the data size, cache size, and cache line size is essential [Coleman and McKinley 1995; Lam et al. 1991]. Higher degrees of tiling can be applied to exploit multilevel caches, the TLB, etc.

(3) Promote register reuse through *unroll-and-jam* (also known as register tiling) and *scalar replacement* [Callahan et al. 1990; Carr and Kennedy 1994a]. 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 article, 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 [Carr 1992]. However, the other steps and interactions between steps are beyond the scope of this article.

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 article extends previous work [Kennedy and McKinley 1992] 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. To achieve this loop structure, we use a compound loop transformation algorithm that consists of loop permutation, fusion, distribution, and reversal. The algorithm 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 nonnegligible, we show there are usually 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. In addition, loop fusion and distribution can produce significant improvements. Our algorithms never found an opportunity where loop reversal could improve locality.

2. BACKGROUND

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

2.1 Data Dependence

We assume the reader is familiar with concept of data dependence [Kuck et al. 1981; Goff et al. 1991]. $\vec{\delta} = \{\delta_1 \ldots \delta_k\}$ 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.

2.2 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 [Wolf and Lam 1991]. Without loss of generality, we assume Fortran's column-major storage.

Since processor speeds outpace memory by factors ranging from 10 to 100 in current uniprocessors, even a single miss in the cache on an inner loop iteration can degrade performance. Our measure of locality is the number of cache lines a loop nest accesses. We minimize accesses to memory by minimizing the number of times a cache line must be fetched from memory.

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

[1] Lam et al. [1991] support this assumption.

```
DO K = 2,N–1
  DO J = 2,N–1
    DO I = 2,N–1
      A(I,J,K) = A(I+1,J+1,K) + B(I,J,K) + B(I,J+1,K) + B(I+1,J,K)
```

Reference Groups

for loop J:	for loop I & K:
{A(I,J,K)}	{A(I,J,K)}
{A(I+1,J+1,K)}	{A(I+1,J+1,K)}
{B(I,J,K), B(I,J+1,K), B(I+1,J,K)}	{B(I,J,K), B(I+1,J,K)}
	{B(I,J+1,K)}

Fig. 1. RefGroup example.

across disjoint nests; it also drives permutation, fusion, distribution, and reversal to improve data locality, thus minimizing the number of cache lines accessed.

2.3 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 [Kennedy and McKinley 1992], but slightly more restrictive than *uniformly generated references* [Gannon et al. 1988]. The goal of the *RefGroup* algorithm is to avoid overcounting cache lines accessed by multiple references that generally access the same set of cache lines.

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. Note that a reference can be in only one reference group, since algorithm *RefGroup* puts a reference in a group if it meets either Conditions (1) or (2) with any other reference in the group. We specify $|d| \leq 2$ in our implementation because previous work on dependence testing found few constant distances greater than 2 [Goff et al. 1991]. In addition, given a cache line size of at least 2 elements and $|d| \leq 2$, the references will only require at most 2 cache lines.

Consider the example nest in Figure 1. Because the two references to A fail all the tests, regardless of the loop, *RefGroup* always places them in distinct groups. For the J loop, B(I,J,K) and B(I,J+1,K) satisfy condition (1b), and B(I,J,K) and

INPUT:

$$\begin{array}{rcl}
\mathcal{L} & = & \{l_1, \ldots, l_n\} \text{ a loop nest with headers } lb_l, ub_l, \; step_l \\
\mathcal{R} & = & \{\; Ref_1, \ldots, \; Ref_m\} \text{ representatives from each reference group} \\
trip_l & = & (ub_l - lb_l + step_l)/step_l \\
cls & = & \text{the cache line size in data items,} \\
coeff(f, i_l) & = & \text{the coefficient of the index variable } i_l \text{ in the subscript } f \\
stride(f_1, i_l, l) & = & |\,\text{step}_l * coeff(f_1, i_l)|
\end{array}$$

OUTPUT:

$$LoopCost(l) \;=\; \text{number of cache lines accessed with } l \text{ as innermost loop}$$

ALGORITHM:

$$\mathbf{LoopCost}(l) = \sum_{k=1}^{m} (\mathbf{RefCost}(Ref_k(f_1(i_1, \ldots, i_n), \ldots, f_j(i_1, \ldots, i_n)), l)) \prod_{h \neq l} trip_h$$

$$\mathbf{RefCost}(Ref_k,\, l) = \begin{cases} 1 & \textit{if } ((coeff(f_1, i_l) = 0) \wedge \ldots \wedge (coeff(f_j, i_l) = 0)) & \textbf{Invariant} \\ \dfrac{trip_l}{\left(\dfrac{\text{cls}}{stride(f_1, i_l, l)}\right)} & \textit{if } ((stride(f_1, i_l, l) < cls) \wedge (coeff(f_2, i_l) = 0) \wedge \ldots \wedge (coeff(f_j, i_l) = 0)) & \textbf{Unit} \\ trip_l & \textit{otherwise} & \textbf{None} \end{cases}$$

Fig. 2. LoopCost algorithm.

B(I+1,J,K) satisfy condition (2). Thus for the J loop, all three references to B are in the same group, even though B(I,J+1,K) and B(I,J+1,K) do not satisfy any of the conditions. Since the I and K loops do not carry the dependence between B(I,J,K) and B(I,J+1,K), only B(I,J,K) and B(I+1,J,K) belong to the same group for the I and K loops.

2.4 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 2. 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.

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 of all the remaining loops. *RefCost* and *LoopCost* appear in Figure 2. This method evaluates imperfectly nested loops (see Section 3.5.1 for an example), complicated subscript expressions, and nests with symbolic bounds [McKinley 1992].

In Figure 3, we give an example of computing *LoopCost* on matrix multiply. Algorithm *RefGroup*, with respect to all of the three loops, puts both references

```
{ 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	$2n^3 + n^2$	$\frac{5}{4}n^3 + n^2$	$\frac{1}{2}n^3 + n^2$

Fig. 3. Loop cost for matrix multiply.

to C(I,J) in one reference group, and A(I,K) and B(K,J) each in their own reference group. *RefCost* with respect to the I loop detects the self-spatial reuse carried by C(I,J) and A(I,K) and assigns each reference the cost of $(1/cls)n$ cache lines. B(K,J) has loop-invariant reuse and a cost of 1. *LoopCost* for the I loop is thus $(1/2)n^3+n^2$ for a machine with $cls = 4$ and n^2 outer iterations of the J and K loops. *LoopCost* with respect to the J and K loops is similar.

3. 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 that determine when individual transformations are profitable. Using these components, Section 3.5 presents *Compound*, an algorithm for discovering and applying legal compound loop nest transformations that aim to minimize the number of cache lines accessed. All of these transformations are implemented in our experimental compiler.

3.1 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 for the innermost loop, l will likely 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* in Figure 4 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. (By definition,

[2]In Section 3.5, we perform imperfect interchanges with distribution. The evaluation method can also drive imperfect loop interchange [Wolfe 1986], but we did not implement it.

INPUT:
$\mathcal{O}$ = $\{i_1, i_2, ..., i_n\}$, the original loop ordering
$\mathcal{DV}$ = set of original legal direction vectors for l_n
$\mathcal{L}$ = $\{i_{\sigma_1}, i_{\sigma_2}, \ldots, i_{\sigma_n}\}$, a permutation of $\mathcal{O}$ with the best estimated locality

OUTPUT:
$\mathcal{P}$ = $\mathcal{L}$, or a permutation of $\mathcal{O}$ that is legally as close to $\mathcal{L}$ as possible

ALGORITHM:
```
if ∀ DV, L is a legal permutation
   return L
P = ∅ ;  k = 0 ;  m = n
while L ≠ ∅
   for j = 1, m
      l = l_j ∈ L
      if direction vectors for {p_1, ..., p_k, l} are legal
         P = {p_1, ..., p_k, l}
         L = L − {l} ;  k = k + 1 ;  m = m − 1
         break for
      endif
   endfor
endwhile
```

Fig. 4. Permute algorithm.

the original distance/direction vector is *legal*, i.e., lexicographically positive [Allen and Kennedy 1984; Banerjee 1990].) 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 is likely to yield the best data locality.

More formally stated, given a memory ordering $\mathcal{L} = \{i_{\sigma_1}, i_{\sigma_2}, \ldots, i_{\sigma_n}\}$ of the loops $\{i_1, i_2, ..., i_n\}$ where i_{σ_1} has the least reuse and i_{σ_n} the most, the algorithm builds up a legal permutation in $\mathcal{P}$ by first testing to see if the loop i_{σ_1} is legal in the outermost position. If it is legal, it is added to $\mathcal{P}$ and removed from $\mathcal{L}$. If it is not legal, the next loop in $\mathcal{L}$ is tested. Once a loop l is positioned, the process is repeated starting from the beginning of $\mathcal{L} - \{l\}$ until $\mathcal{L}$ is empty.

Permute works by positioning the outer k loops such that the partial direction vectors are lexicographically positive, which means either all entries are zero, or at least one positive entry precedes negative entries in all the partial direction vectors $\{p_1, \ldots, p_k\}$. Consider placing a loop l at position $k+1$ for a single dependence. If the direction vector entry at position l is positive or zero, l is legal in position $k+1$. If the entry at l is negative and $\{p_1, \ldots, p_k\}$ positive, l is also legal in position $k+1$. However, if the entry at position l is negative, and $\{p_1, \ldots, p_k\}$ is zero, then positioning l would create a negative, illegal direction vector. Notice that no permutation of $\{p_1, \ldots, p_k\}$ can change a positive vector to zero and thus enable a negative entry at l to be placed in position $k+1$. This property enables *Permute* to work greedily from the outermost loop to the innermost. The following theorem holds for the *Permute* algorithm.

THEOREM. *If there exists a legal permutation where σ_n is the innermost loop, then Permute will find a permutation where σ_n is innermost.*

Given an original set of legal direction vectors, if $\mathcal{L}$ is legal then σ_n is clearly innermost. Otherwise, the proof by contradiction of the theorem proceeds as follows. Each step of the "for" is guaranteed to find a loop that results in partial, positive direction vectors; otherwise the original was not legal [Allen and Kennedy 1984; Banerjee 1990]. In addition, if any loop σ_1 through σ_{n-1} may be legally positioned prior to σ_n, it will be.

Permute therefore places the loops carrying the most reuse as innermost as possible. If the desired inner loop cannot be obtained, it places the next most desirable inner loop in the innermost position if possible, and so on. This characteristic is important because most data reuse occurs on the innermost loop, so positioning it correctly is key to achieving the best data locality.

Complexity. When memory order is legal, as it is in 80% of the loops in our test suite, *Permute* simply sorts loops according to their LoopCost and tests for legality. Otherwise algorithm *Permute* selects a legal permutation as close to memory order as possible, testing the legality of $n(n-1)$ loop permutations in the worst case. However, these steps only involve testing data dependences; evaluating the locality of the loop nest turns out to be the most expensive part of the algorithm. Our algorithm computes the best permutation with one evaluation step (i.e., invocation of *LoopCost*) for each loop in the nest. The complexity of algorithm *Permute* is therefore $O(n)$ in the number of *LoopCost* invocations, where n is the number of loops in the nest.

3.1.1 *Example: Matrix Multiplication.* In Figure 3, we saw that 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 5. 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 × 512 versus the 300 × 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.

3.2 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 lexicograph-

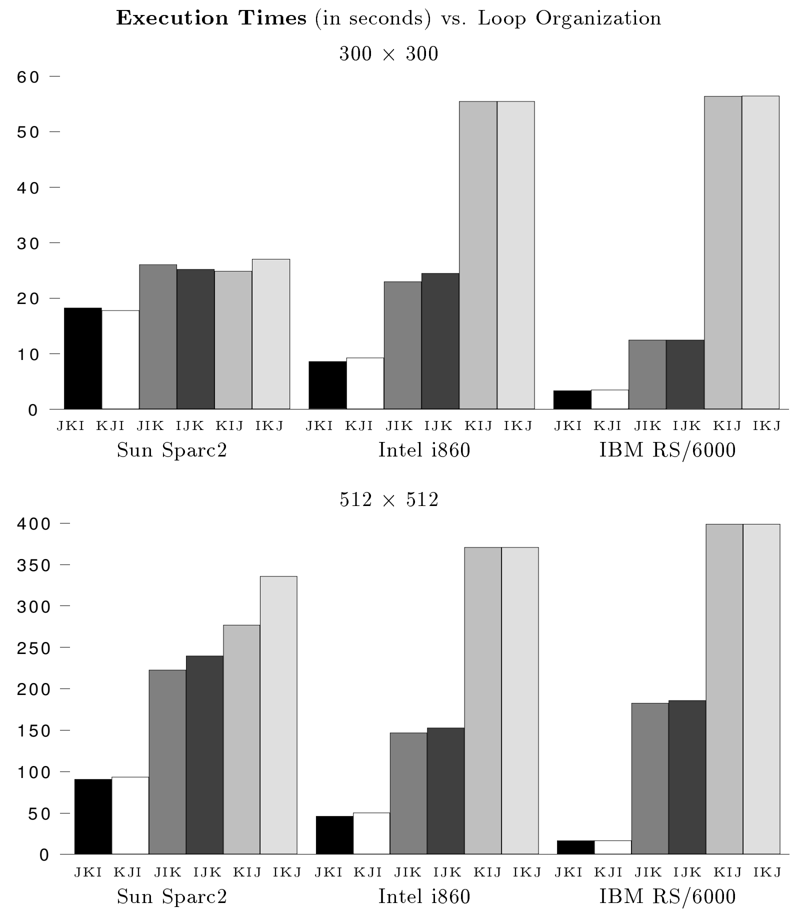

Fig. 5. Performance of matrix multiply.

ically positive dependence vectors. 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 improve locality in our experiments; therefore we will not discuss it further.

3.3 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 [Warren 1984]. As an example of its effect, consider the code fragment written in Fortran 90 in Figure 6(a) that performs ADI integration. *Scalarizing* the Fortran 90 into Fortran 77 results in the code in Figure 6(b) which 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,

(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*	$2 * n^2$	$\frac{1}{2} * n^2$
$S_1 + S_2$	$5 * n^2$	$\frac{5}{4} * n^2$

Fig. 6. Loop fusion.

significantly improving spatial locality for all the arrays. This transformation is illustrated in Figure 6(c).

3.3.1 *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 if 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:

(1) Compute *RefGroup* and *LoopCost* as if all the statements were in the same nest, i.e., fused.

(2) Compute *RefGroup* and *LoopCost* independently for each candidate and add the results.

(3) Compare the total *LoopCosts*.

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

Fig. 7. Fusion algorithm.

If the fused *LoopCost* is lower, fusion alone will result in additional locality. For example, fusing the two K loops in Figure 6 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.

3.3.2 *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 6 enables permutation of 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 if it creates a perfect nest in which memory order can be achieved.

3.3.3 *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 m compatible loop nests is NP-hard [Kennedy and McKinley 1993]. In this work, the problem is harder, since 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 *LoopCosts* 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 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 7.

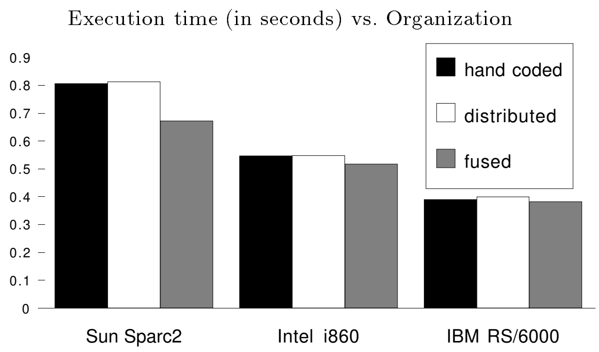

Fig. 8. Performance of Erlebacher.

Since we allow loop nests to be reordered due to fusion, we may need to calculate *LoopCost* for every pair of loop nests. The complexity of the fusion algorithm is therefore $O(m^2)$ in the number of invocations of *LoopCost*, where m is the number of candidate nests for fusion. If we only fused adjacent loop nests, the complexity of the algorithm would drop to $O(m)$.

3.3.4 *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 applied loop distribution by hand to loops containing multiple statements, placing each statement in a separate loop nest. Since the loops are fully distributed in this version of Erlebacher, it resembles the output of a Fortran 90 scalarizer.

From the fully distributed version of Erlebacher, we created two optimized versions of the program. In the first, we applied *Permute* to transform individual loop nests into memory order. In the second optimized version, we applied *Fuse* to obtain more temporal locality. In Figure 8, we measure the performance of the original program (Hand), the transformed fully distributed program (Distributed), and the fused version (Fused).

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. Its impact will only increase as more programs are written with Fortran 90 array syntax.

3.4 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 that does not include input dependences) must be placed in the same loop. Groups of statements which must be in the same loop are called *partitions*. In our system we only use loop distri-

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 S into finest partitions P = {p_1, ..., p_m}
      s.t. if s_r, s_t ∈ a recurrence, s_r, s_t ∈ p_i.
    compute MemoryOrder_i for each p_i
    if ( ∃ i | MemoryOrder_i is achievable with
      distribution and permutation)
        perform distribution and permutation
        return
endfor
```

Fig. 9. Distribution algorithm.

bution 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 9. It divides the statements into the finest granularity partitions and tests if that enables loop permutation.[4] It performs distribution on the innermost loop that 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.

We only invoke algorithm *Distribute* if memory order cannot be achieved on a nest and if not all of the inner nests can be fused (see Section 3.5). *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*. Since *LoopCost* is calculated for each individual partition, the complexity of algorithm *Distribute* is $O(m)$, where m is the number of individual partitions created by loop distribution. See Section 3.5.1 for an example.

3.5 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 10 considers adjacent loop nests. It first optimizes each nest independently and then applies fusion between the resulting nests

[3]Distribution could also be effective if (1) there is no temporal locality between partitions, and the accessed arrays are too numerous to fit in cache at once, or (2) register pressure is a concern. We do not address these issues here.

[4]The compound transformation algorithm in Section 3.5 follows distribution and permutation with fusion to regain lost temporal locality.

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(N)
```

Fig. 10. Compound loop transformation algorithm.

if legal, and data 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. The complexity of algorithm *Compound* is $O(nm^2)$ in the number of invocations of *LoopCost*, where n is the number of loops in a nest and m the number of adjacent loop nests. $O(n)$ invocations of *LoopCost* are needed to calculate memory order for each loop nest, and the process may need to be repeated up to $O(m^2)$ times when applying loop fusion. Fortunately, fusion and distribution only need to be invoked if the original loop nest cannot be permuted into memory order. In practice, the loop fusion algorithm is seldomly applied and does not need to consider many adjacent loop nests. Loop distribution may increase m, the number of adjacent loop nests, by creating additional loop nests. In the worst case it can increase m to the number of statements in the program. The increase in the number of loop nests was negligible in practice; a single application of distribution never created more than three new nests.

Compilation Time. Accurately estimating the increase in compilation time caused by applying algorithm *Compound* is difficult. First, our implementation depends upon the efficiency of the ParaScope infrastructure [Cooper et al. 1993]. Second, our implementation on top of ParaScope is not especially efficient. Given these two caveats, our tests showed a 25% increase in compilation time over just parsing and dependence analysis when *Compound* is applied. The time required for algorithm *Compound* is only 33% of the time required to apply dependence analysis alone.

We feel that this cost is not prohibitive for highly optimizing compilers.

3.5.1 *Example: Cholesky Factorization.* Consider optimizing the Cholesky Factorization kernel in Figure 11(a). Notice that there are references to A(K,K) nested at different levels. Since these references have temporal locality, *RefGroup* places them in the same group. *LoopCost* then uses the most deeply nested reference to compute the cost in cache lines of A(K,K). For the entire nest, *LoopCost* selects KJI as the best loop organization and ranks the nests from lowest cost to highest (KJI, JKI, KIJ, IKJ, JIK, IJK). *Compound* then tries to achieve this loop organization.

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 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 11(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 [Wolfe 1991].) 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.

4. 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. We measured execution times on two architectures: the IBM RS/6000 model 540 and the HP PA-RISC model 715/50. To measure our ability to improve locality, we also determined (for our memory model) the best locality achievable through loop transformations in the *ideal case*, assuming correctness could be ignored. We collected statistics on the data locality in the original, transformed, and *ideal* programs. These statistics use the cache configuration of the IBM RS/6000.

4.1 Methodology

We implemented the cost model, the transformations, and the algorithms described above in Memoria, the memory compiler in the ParaScope Programming Environment [Carr 1992; Carr and Kennedy 1994b; Cooper et al. 1993; Kennedy et al. 1993]. 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, we perform auxiliary induction variable substitution, constant propagation, forward expression propagation, and dead-code elimination using PFC [Allen and Kennedy 1987].[5] Memoria also determines if scalar expansion will further enable distribution. Since scalar expansion is not integrated in

[5]Note that for our execution-time experiments on the HP PA-RISC, we were only able to perform dependence analysis on the codes because PFC lost its platform (PFC runs on an IBM 370 and is written in PL/I.)

(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) ⇓ {KJI form} *Loop Distribution & Triangular Interchange* ⇓

```
        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*	$2n^3+n^2$	$\frac{5}{4}n^3+n^2$	$\frac{1}{2}n^3+n^2$

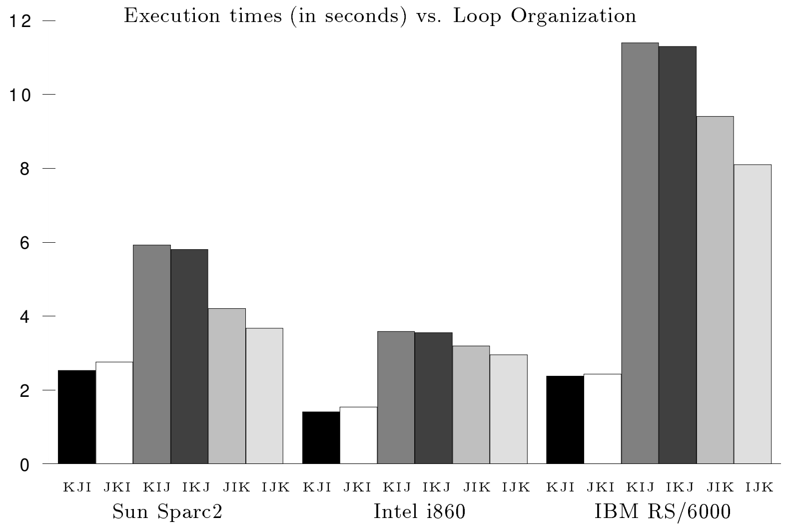

Fig. 11. Cholesky factorization.

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 noncomment lines. Their execution times on the IBM RS/6000 ranged from seconds to a couple of hours.

4.2 Transformation Results

In Table I, we report the results of transforming the loop nests of each program. For each program, Table I first lists the number of loop nests (N) of depth 2 or more which were considered for transformation. **Mem Order** and **Inner Loop** columns reflect the percentage of loop nests and inner loops, respectively, that are:

— **O**: originally in memory order,
— **P**: permuted into memory order, or
— **F**: fail to achieve memory order.

These three numbers sum to 100%. 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 percentage of nests where the most desirable innermost loop is positioned correctly.

Table I 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,
— **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 Dist** column,

— **D** is the number of loop nests distributed to achieve better loop permutation,
— **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*, six or more nests resulted.

LoopCost Ratio in Table I estimates the potential reduction in LoopCost for the final transformed program (**F**) and the *ideal* program (**I**) over the entire program. Remember that the ideal program achieves memory order for every nest without

Table I. Memory Order Statistics

Prog	N	Mem Order O	Mem Order P	Mem Order F	Inner Loop O	Inner Loop P	Inner Loop F	Loop Fusion C	Loop Fusion A	Loop Dist D	Loop Dist R	LoopCost Ratio F	LoopCost Ratio I
		%	percentages	%									
Perfect Benchmarks													
adm	106	52	16	32	53	16	31	0	0	1	2	2.5	6.1
arc2d	75	55	28	17	65	34	1	35	12	1	2	2.2	4.1
bdna	56	75	18	7	75	18	7	4	2	3	6	2.3	2.5
dyfsm	80	63	15	22	65	19	16	2	1	0	0	3.0	8.6
flo52	76	83	17	0	95	5	0	4	1	0	0	1.7	1.7
mdg	12	83	8	8	83	8	8	0	0	0	0	1.1	1.7
mg3d	40	95	3	3	98	0	2	0	0	1	2	1.0	1.1
ocean	56	82	13	5	84	13	4	2	1	3	6	2.0	2.2
qcd	45	53	11	36	58	16	15	0	0	0	0	4.9	6.1
spc77	162	64	7	29	66	7	27	0	0	0	0	2.3	5.5
track	32	50	16	34	56	19	25	2	1	1	2	1.9	7.9
trfd	29	52	0	48	66	0	34	0	0	0	0	1.0	15
SPEC Benchmarks													
dnsa7	50	64	14	22	74	16	10	5	2	1	2	2.0	2.9
doduc	33	6	6	88	6	6	88	0	0	4	12	1.8	14
fpppp	8	88	12	0	88	12	0	0	0	0	0	1.0	1.0
hyd2d	55	100	0	0	100	0	0	44	11	0	0	1.0	1.0
m300	2	50	50	0	50	50	0	0	0	1	2	4.5	4.5
mdp2	1	0	0	100	0	0	100	0	0	0	0	1.0	1.0
msp2	1	0	0	100	0	0	100	0	0	0	0	1.0	1.0
ora	3	100	0	0	100	0	0	0	0	0	0	1.0	1.0
sucor	36	42	19	39	42	19	39	0	0	4	8	3.5	5.3
s256	8	88	12	0	88	12	0	0	0	0	0	4.9	4.9
tcatv	6	100	0	0	100	0	0	7	2	0	0	1.0	1.0
NAS Benchmarks													
appbt	87	98	0	2	100	0	0	3	1	0	0	1.0	1.2
applu	71	73	3	24	79	6	15	3	1	2	6	1.3	8.0
appsp	84	73	12	15	80	12	8	8	4	0	0	1.2	4.3
buk	0	0	0	0	0	0	0	0	0	0	0	1.0	1.0
cgm	6	0	0	100	0	0	100	0	0	0	0	1.0	2.7
embar	2	50	0	50	50	0	50	0	0	0	0	1.0	1.1
fftpd	18	89	0	11	100	0	0	0	0	0	0	1.0	1.0
mgrid	19	89	11	0	100	0	0	3	1	1	2	1.0	1.0
Miscellaneous Programs													
erle	30	83	13	4	100	0	0	28	11	0	0	1.0	1.0
lpckd	4	75	0	25	75	0	25	3	1	0	0	1.0	1.1
simpl	22	86	9	5	86	9	5	6	2	0	0	2.4	2.7
wave	85	58	29	13	65	29	6	70	26	0	0	4.2	4.3
total	1400	69	11	20	74	11	15	229	80	23	52	—	—

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. This ratio includes loops that *Compound* did not transform and reveals the potential for locality improvement.

Memoria may not obtain memory order due to the following reasons: (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 tests may enable the algorithms to transform a few more nests.

4.3 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 [Cooper et al. 1993; Hall et al. 1991]; 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 of 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.

4.4 Successful Transformation

We illustrate our ability to transform for data locality by program in Figures 13 and 12. The figures characterize the programs by the percentage of their nests and inner loops that are originally in memory order and that we transform into memory order. In Figure 13, half of the original programs have fewer than 70% of their nests in memory order. In the transformed versions, 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 12 are more dramatic. The majority of the programs now have 90% or more of their inner loops positioned correctly for the best locality (according to our memory model). Our transformation algorithms can thus 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 these benchmark programs tend to be small enough to fit in cache; the transformed loop nests may be CPU

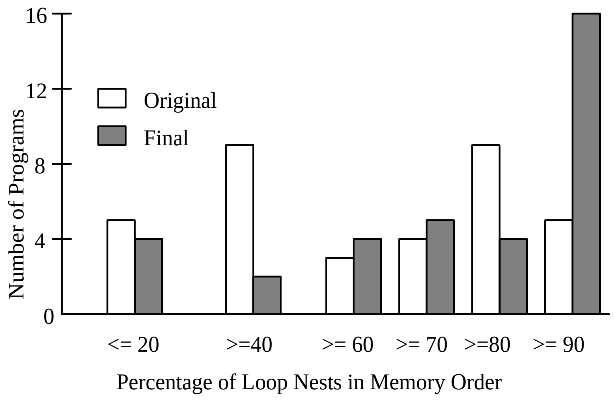

Fig. 12. Achieving memory order for the inner loop.

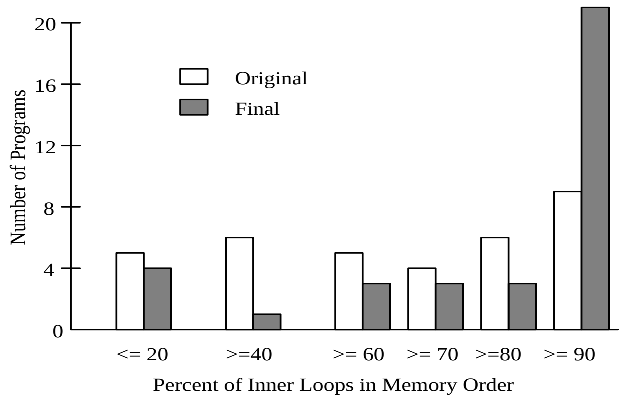

Fig. 13. Achieving memory order for loop nests.

bound instead of memory bound; and the optimized portions of the program may not significantly contribute to the overall execution time.

4.5 Performance Results

In Figure 14, 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. In Figure 15, we present the performance of our test suite on an HP 715/50 with a 64KB direct-mapped cache with 32-byte cache lines. Figures 14 and 15 present detailed results for four kernels from *Dnasa*: *Btrix, Emit, Gmtry,* and *Vpenta*. Results are reported in normalized execution time with the base time of 100 not indicated. The arithmetic mean in each figure includes only those programs shown in the bar graph. On both machines, we used the standard 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. Applications *Flo52* and *Wave* did not compile and run on the HP. For those applications not

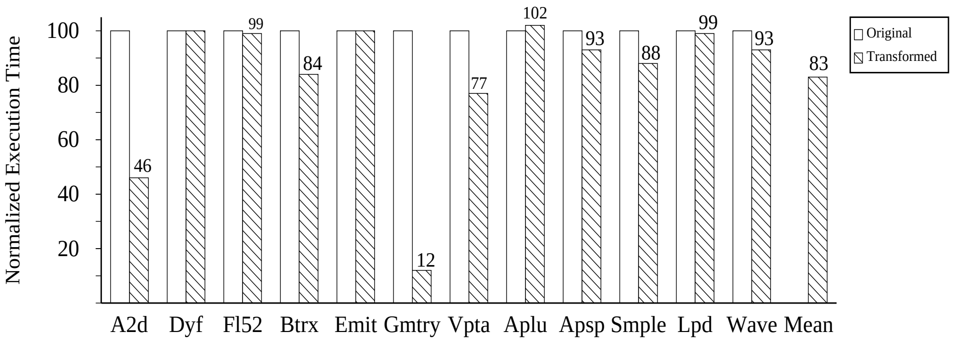

Fig. 14. Performance results on IBM RS/6000 model 540.

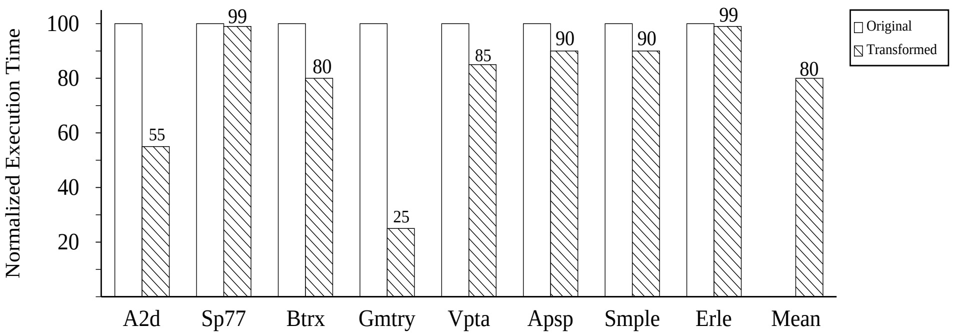

Fig. 15. Performance results on HP 715/50.

listed in Figure 14 and Figure 15, no performance improvement or degradation occurred.

Figure 14 and Figure 15 show a number of applications with significant performance improvements: *Arc2d*, *Dnasa7* (*Btrix, Emit, Gmtry, Vpenta*), *Appsp*, 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 our 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).[6] The i860 cache was chosen to reveal the potential of our optimizations on a small cache. For each program and cache, we determined the change in the hit rates both for just the optimized procedures and for the entire program. Table II presents these rates. Small variations in cache hit rates after program transformations can be caused by changes in cache interference and code generation. Places where the compiler affected cache hit rates by $\geq 0.1\%$ are emboldened for greater emphasis.

[6]Carr and Wu [1995] also simulate an HP-style cache, but their results are similar to the RS/6000.

For the *Final* columns we chose the better of the fused and unfused versions for each program.

As illustrated in Table II, 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 8KB, 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. The improvements did not always carry over to the entire program, since the unoptimized nests may still dominate the execution time.

We measured hit ratios both with and without applying loop fusion. For the 8KB 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 in *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 [Coleman and McKinley 1995; Lam et al. 1991]. 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 [Wolf 1992]. 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% 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.

4.6 Data Access Properties

To further interpret our results, we measured the data access properties for our test suite. We report the data access properties for the inner loops on the original (orig), ideal memory order, and final versions of the programs in Tables III and IV. **Locality of Reference Group** classifies the percentage of RefGroups displaying

Table II. Simulated Cache Hit Rates

Program	Optimized Procedures				Whole Program			
	Cache 1		Cache 2		Cache 1		Cache 2	
	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**

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

each form of self-reuse as invariant (I), unit-stride (U), or none (N). (G) 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

Table III. Data Access Properties

Program		Locality of Reference Groups: % Groups I	U	N	Gp	Refs/Group I	U	N	Avg	LoopCost Ratios Avg	Wt
Perfect Benchmarks											
adm	*orig*	4	70	26	0	1.04	1.39	1.34	1.36		
	final	5	83	12	0	1.03	1.38	1.32	1.36	2.54	2.68
	ideal	19	77	4	0	1.50	1.32	1.10	1.34	6.10	6.24
arc2d	*orig*	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
bdna	*orig*	2	62	36	0	2.00	1.08	1.04	1.08		
	final	2	64	34	0	2.00	1.08	1.03	1.08	2.31	2.24
	ideal	5	61	34	0	1.52	1.07	1.03	1.08	2.51	2.44
dyfesm	*orig*	8	55	37	0	1.19	1.20	1.25	1.21		
	final	12	61	27	0	1.44	1.15	1.25	1.21	3.08	3.06
	ideal	22	60	18	0	1.46	1.17	1.05	1.21	8.62	9.93
flo52	*orig*	1	92	7	0	1.50	1.38	1.00	1.35		
	final	1	94	5	0	1.50	1.37	1.00	1.35	1.72	1.79
	ideal	1	94	5	0	1.50	1.37	1.00	1.35	1.72	1.79
mdg	*orig*	1	75	24	0	2.00	1.14	1.00	1.12		
	final	0	76	24	0	0	1.16	1.00	1.12	1.11	1.09
	ideal	1	78	21	0	1.00	1.15	1.00	1.12	1.70	1.63
mg3d	*orig*	0	4	96	0	0	1.26	1.00	1.01		
	final	0	4	96	0	0	1.26	1.00	1.01	1.00	1.00
	ideal	0	4	96	0	1.00	1.27	1.00	1.01	1.13	1.12
ocean	*orig*	0	56	44	0	0	1.07	1.00	1.04		
	final	0	69	31	0	0	1.06	1.00	1.04	2.05	2.16
	ideal	2	67	31	0	1.33	1.05	1.00	1.04	2.20	2.30
qcd	*orig*	34	42	24	0	2.27	1.22	1.53	1.65		
	final	43	47	10	0	2.03	1.28	1.75	1.65	3.71	3.73
	ideal	51	40	9	0	2.05	1.10	1.86	1.65	6.40	6.65
spec77	*orig*	5	42	53	0	1.57	1.56	1.37	1.46		
	final	10	43	47	0	3.00	1.58	1.04	1.46	3.22	3.10
	ideal	25	33	42	0	2.00	1.59	1.00	1.45	5.59	5.60
track	*orig*	7	75	18	0	1.40	1.09	1.23	1.14		
	final	7	81	12	0	1.20	1.15	1.00	1.14	1.99	1.84
	ideal	36	60	4	0	1.19	1.11	1.00	1.14	7.95	9.68
trfd	*orig*	7	62	31	2	1.50	1.28	1.00	1.21		
	final	7	62	31	2	1.50	1.28	1.00	1.21	1.00	1.00
	ideal	52	34	14	2	1.40	1.00	1.00	1.21	14.81	17.34
SPEC Benchmarks											
dnasa7	*orig*	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
doduc	*orig*	10	2	88	0	1.24	1.33	1.17	1.18		
	final	7	63	30	0	1.00	1.29	1.00	1.18	5.44	5.44
	ideal	7	64	29	0	1.00	1.28	1.00	1.18	5.45	5.45
fpppp	*orig*	0	4	96	0	0	1.00	1.00	1.00		
	final	0	5	95	0	0	1.00	1.00	1.00	1.03	1.03
	ideal	0	5	95	0	0	1.00	1.00	1.00	1.03	1.03
matrix300	*orig*	0	75	25	0	0	1.00	1.00	1.00		
	final	0	100	0	0	0	1.00	0	1.00	4.50	4.50
	ideal	0	100	0	0	0	1.00	0	1.00	4.50	4.50
tomcatv	*orig*	2	70	28	0	1.00	1.24	1.00	1.17		
	final	2	70	28	0	1.00	1.24	1.00	1.17	1.00	1.00
	ideal	2	70	28	0	1.00	1.24	1.00	1.17	1.00	1.00

Table IV. Data Access Properties

Program		Locality of Reference Groups: % Groups I	U	N	G	Refs/Group I	U	N	Avg	LoopCost Ratios Avg	Wt
NAS Benchmarks											
appbt	*orig*	0	17	83	0	0	1.04	1.00	1.01		
	final	0	17	83	0	0	1.04	1.00	1.01	1.00	1.00
	ideal	0	17	83	0	1.67	1.03	1.00	1.01	1.26	1.38
applu	*orig*	0	26	74	0	2.00	1.05	1.06	1.06		
	final	1	27	72	0	1.25	1.06	1.06	1.06	1.35	1.50
	ideal	8	23	69	0	1.45	1.07	1.01	1.06	8.03	10.06
appsp	*orig*	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
buk	*orig*	0	0	0	0	0	0	0	0	0	
	final	0	0	0	0	0	0	0	0	1.00	1.00
	ideal	0	0	0	0	0	0	0	0	1.00	1.00
cgm	*orig*	0	38	62	0	0	1.10	1.00	1.04		
	final	0	38	62	0	0	1.10	1.00	1.04	1.00	1.00
	ideal	38	0	62	0	1.10	0	1.00	1.04	2.75	2.62
embar	*orig*	0	50	50	0	0	1.00	1.00	1.00		
	final	0	50	50	0	0	1.00	1.00	1.00	1.00	1.00
	ideal	50	0	50	0	1.00	0	1.00	1.00	1.12	1.12
fftpde	*orig*	0	72	28	0	0	1.02	1.00	1.01		
	final	0	72	28	0	0	1.02	1.00	1.01	1.00	1.00
	ideal	0	72	28	0	0	1.02	1.00	1.01	1.00	1.00
mgrid	*orig*	15	56	29	0	1.12	1.97	1.00	1.56		
	final	15	56	29	0	1.12	1.97	1.00	1.56	1.00	1.00
	ideal	15	56	29	0	1.12	1.97	1.00	1.56	1.00	1.00
Miscellaneous Programs											
erlebacher	*orig*	23	82	20	0	1.22	1.52	1.55	147		
	final	23	82	20	0	1.22	1.52	1.55	147	1.00	1.00
	ideal	23	82	20	0	1.22	1.52	1.55	147	1.00	1.00
linpackd	*orig*	0	55	45	0	0	1.00	1.05	1.02		
	final	0	55	45	0	0	1.00	1.05	1.02	1.00	1.00
	ideal	0	57	43	0	0	1.04	1.00	1.02	1.10	1.10
simple	*orig*	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	*orig*	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	*orig*	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	—	—

LoopCost Ratio column estimates the potential improvement as an average (Avg) over all the nests, and a weighted average (Wt) uses nesting depth. The last row contains the totals for all the programs.

Table III reveals that each of the applications we improved (*Arc2d*, *Dnasa7*, *Appsp*, *Simple*, and *Wave*) had a significant gain in self-spatial reuse (Unit) on the inner loop over the original program. Spatial locality was the key to getting good cache performance. Although programmers can make the effort to ensure unit-stride access in their 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 IV indicates 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. For many programs (e.g., *Adm, Trfd, Dnasa7, Embar*), 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. Our analysis usually determines that spatial reuse is of more benefit than temporal reuse when they are carried on different loops. In some cases, tiling may be able to exploit invariant reuse carried by outer loops and continue to benefit from the spatial reuse carried by inner loops.

4.7 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 and HP PA-RISC.

Arc2d is a fluid-flow solver from the Perfect Benchmarks. The main computational routines exhibit poor cache performance due to nonunit 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 Figure 14 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 Figure 14. 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 [Callahan et al. 1988; Carr and Kennedy 1994a]. 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 structure may have been how the author viewed Gaussian elimination conceptually, it translated to poor performance. Distribution and permutation achieved 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 only on the RS/6000 (2%). 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 in the inner loop better performance on the RS/6000. Locality within the two innermost loops is not a problem.

5. RELATED WORK

Abu-Sufah [1979] first discussed applying compiler transformations based on data dependence (e.g., loop interchange, fusion, distribution, and tiling) to improve paging. In this article, we extend and validate recent research to integrate optimizations that target parallelism and the memory hierarchy [Kennedy and McKinley 1992]. We extend the 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 and present extensive experimental results.

Our approach has several advantages over previous research. 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 work is applicable to a wider range of programs because we do not require perfect nests or nests that can be made perfect with conditionals [Ferrante et al. 1991; Gannon et al. 1988; Li and Pingali 1992; Wolf and Lam 1991]. It is also quicker, both in the expected and worse case.

Previous research focused on evaluating data locality when given a loop permutation [Ferrante et al. 1991; Gannon et al. 1988]. Since they must evaluate a given permutation, they may consider up to $n!$ loop permutations (though n is typically small) in order to find the loop permutation which yields the best data locality. (Neither paper specifies an algorithm for generating a smaller search space.) In comparison, our approach evaluates the reuse carried by each loop and directly determines the best loop permutation. Since evaluation is the most expensive step, we expect our algorithm will be much faster in practice. Our algorithm is also the first to combine loop fusion and distribution with loop permutation.

Wolf and Lam [1991] use unimodular transformations (a combination of permutation, skewing, and reversal) and tiling with estimates of temporal and spatial reuse to improve data locality. They prune their search space by ignoring loops that do not carry reuse and loops that cannot be permuted due to legality constraints, but may still have many legal loop organizations remaining whose locality must be evaluated. 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 [Wolf and Lam 1991; Wolf 1992]. 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. However, we improve a few more programs/routines than they do. In addition, their cache optimizations degrade six programs/routines, in one case by 20%. We degrade only one program by a slight 2%: *Applu* from the NAS Benchmarks. In Wolf and Lam's experiments, skewing was never needed, and reversal was seldom applied [Wolf 1992]. We therefore chose not to include skewing, even though (1) it is implemented in our system [Kennedy et al. 1993] and (2) our model can drive it. We did integrate reversal, but it did not help to improve locality.

Li and Pingali [1992] use linear transformations (any linear mapping from one loop nest to another loop nest) to optimize for both data locality and parallelism. They do not propose exhaustive search, since the search space becomes infinite, but transform the loop nest based on certain references in the program. They give no details of their heuristic to order loops for locality. We therefore offer no comparison on effectiveness or complexity.

Applying an exhaustive search approach is not practical when including loop fusion and distribution because they create and combine loop nests. Fusion for improving reuse is by itself NP-hard [Kennedy and McKinley 1993]. By driving heuristics with a cache model, our algorithms are efficient and usually find the best loop organization for data locality using permutation, fusion, and distribution.

When compared with previous work [Gannon et al. 1988; Wolf and Lam 1991], our cache model loses precision in the *RefGroup* and *LoopCost* algorithms because of simplifying assumptions about outer loops. Because our algorithms do not consider the order of outer loops, they miss loop invariance when it spans multiple inner loops. In practice, this inaccuracy does not affect our ability to derive the best loop organization, since the algorithms find and compare invariance and other forms of reuse precisely for innermost loops. If we cannot position the best inner loop, we may miss a better outer loop organization. But this imprecision is exactly what enables us to achieve a single evaluation step and lower algorithmic complexity. It is an open question whether a more precise cache model will yield performance improvements in practice for real applications.

6. TILING

Permuting loops into memory order maximizes estimated short-term cache-line reuse across iterations of inner loops. The compiler can also apply *loop tiling*, a combination of strip-mining and loop interchange, to capture long-term invariant reuse at outer loops [Coleman and McKinley 1995; Irigoin and Triolet 1988; Lam et al. 1991; Wolf and Lam 1991; Wolfe 1987]. 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 nonconsecutive 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 article with tiling, unroll-and-jam, and scalar replacement.

7. CONCLUSION

This article 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 simplifying assumptions used in our model do not appear to hinder the compiler's ability to exploit data locality for scientific applications. The empirical results presented in this article 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, particularly for vector and Fortran 90 programs. We believe this is a significant step toward 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 obliged to Peter Craig at Digital for inspiring the addition of loop reversal. 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. We also wish to thank Qunyan Wu who ran the experiments on the HP 715/50.

REFERENCES

Abu-Sufah, W. 1979. Improving the performance of virtual memory computers. Ph.D. thesis, Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign.

Allen, J. R. and Kennedy, K. 1984. Automatic loop interchange. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction.* ACM, New York.

Allen, J. R. and Kennedy, K. 1987. Automatic translation of Fortran programs to vector form. *ACM Trans. Program. Lang. Syst. 9,* 4 (Oct.), 491–542.

Banerjee, U. 1990. A theory of loop permutations. In *Languages and Compilers for Parallel Computing*, D. Gelernter, A. Nicolau, and D. Padua, Eds. The MIT Press, Cambridge, Mass., 54–74.

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

Callahan, D., Cocke, J., and Kennedy, K. 1988. Estimating interlock and improving balance for pipelined machines. *J. Parall. Distrib. Comput. 5,* 4 (Aug.), 334–358.

Carr, S. 1992. Memory-hierarchy management. Ph.D. thesis, Dept. of Computer Science, Rice Univ., Houston, Tex.

Carr, S. and Kennedy, K. 1994a. Improving the ratio of memory operations to floating-point operations in loops. *ACM Trans. Program. Lang. Syst. 16,* 6 (Nov.), 1769–1810.

Carr, S. and Kennedy, K. 1994b. Scalar replacement in the presence of conditional control flow. *Softw. Prac. Exper. 24,* 1 (Jan.), 51–77.

Carr, S. and Wu, Q. 1995. An analysis of loop permutation on the HP PA-RISC. Tech. Rep. TR95-03, Michigan Technological Univ., Houghton, Mich. Feb.

Coleman, S. and McKinley, K. S. 1995. Tile size selection using cache organization and data layout. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation.* ACM, New York.

COOPER, K., HALL, M. W., HOOD, R. T., KENNEDY, K., MCKINLEY, K. S., MELLOR-CRUMMEY, J. M., TORCZON, L., AND WARREN, S. K. 1993. The ParaScope parallel programming environment. *Proc. IEEE 81,* 2 (Feb.), 244–263.

COOPER, K., HALL, M. W., AND KENNEDY, K. 1993. A methodology for procedure cloning. *Comput. Lang. 19,* 2 (Feb.), 105–117.

FERRANTE, J., SARKAR, V., AND THRASH, W. 1991. On estimating and enhancing cache effectiveness. In *Languages and Compilers for Parallel Computing, 4th International Workshop*, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, Eds. Springer-Verlag, Berlin, 328–343.

GANNON, D., JALBY, W., AND GALLIVAN, K. 1988. Strategies for cache and local memory management by global program transformation. *J. Parall. Distrib. Comput. 5,* 5 (Oct.), 587–616.

GOFF, G., KENNEDY, K., AND TSENG, C.-W. 1991. Practical dependence testing. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation.* ACM, New York.

HALL, M. W., KENNEDY, K., AND MCKINLEY, K. S. 1991. Interprocedural transformations for parallel code generation. In *Proceedings of Supercomputing '91.* IEEE, New York.

IRIGOIN, F. AND TRIOLET, R. 1988. Supernode partitioning. In *Proceedings of the 15th Annual ACM Symposium on the Principles of Programming Languages.* ACM, New York.

KENNEDY, K. AND MCKINLEY, K. S. 1992. Optimizing for parallelism and data locality. In *Proceedings of the 1992 ACM International Conference on Supercomputing.* ACM, New York.

KENNEDY, K. AND MCKINLEY, K. S. 1993. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *Languages and Compilers for Parallel Computing*, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, Eds. Springer-Verlag, Berlin, 301–321.

KENNEDY, K., MCKINLEY, K. S., AND TSENG, C.-W. 1993. Analysis and transformation in an interactive parallel programming tool. *Concurrency Pract. Exper. 5,* 7 (Oct.), 575–602.

KUCK, D., KUHN, R., PADUA, D., LEASURE, B., AND WOLFE, M. J. 1981. Dependence graphs and compiler optimizations. In *Conference Record of the 8th Annual ACM Symposium on the Principles of Programming Languages.* ACM, New York.

LAM, M., ROTHBERG, E., AND WOLF, M. E. 1991. The cache performance and optimizations of blocked algorithms. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems.* ACM, New York.

LI, W. AND PINGALI, K. 1992. Access normalization: Loop restructuring for NUMA compilers. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems.* ACM, New York.

MCKINLEY, K. S. 1992. Automatic and interactive parallelization. Ph.D. thesis, Dept. of Computer Science, Rice Univ., Houston, Tex.

WARREN, J. 1984. A hierachical basis for reordering transformations. In *Conference Record of the 11th Annual ACM Symposium on the Principles of Programming Languages.* ACM, New York.

WOLF, M. E. 1992. Improving locality and parallelism in nested loops. Ph.D. thesis, Dept. of Computer Science, Stanford Univ., Stanford, Calif.

WOLF, M. E. AND LAM, M. 1991. A data locality optimizing algorithm. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation.* ACM, New York.

WOLFE, M. J. 1986. Advanced loop interchanging. In *Proceedings of the 1986 International Conference on Parallel Processing.* CRC Press, Boca Raton, Fla.

WOLFE, M. J. 1987. Iteration space tiling for memory hierarchies. In *Proceedings of the 3rd SIAM Conference on Parallel Processing.* SIAM, Philadelphia, Pa.

WOLFE, M. J. 1991. The Tiny loop restructuring research tool. In *Proceedings of the 1991 International Conference on Parallel Processing.* CRC Press, Boca Raton, Fla.

Received August 1995; revised January 1996; accepted March 1996