Optimizing for Parallelism and Data Locality[*]

Ken Kennedy   Kathryn S. McKinley

ken@rice.edu   *kats@rice.edu*

Department of Computer Science
Rice University
Houston, TX 77251-1892

Abstract

Previous research has used program transformation to introduce parallelism and to exploit data locality. Unfortunately, these two objectives have usually been considered independently. This work explores the trade-offs between effectively utilizing parallelism and memory hierarchy on shared-memory multiprocessors. We present a simple, but surprisingly accurate, memory model to determine cache line reuse from both multiple accesses to the same memory location and from consecutive memory access. The model is used in memory optimizing and loop parallelization algorithms that effectively exploit data locality and parallelism in concert. We demonstrate the efficacy of this approach with very encouraging experimental results.

1 Introduction

Transformations to exploit parallelism and to improve data locality are two of the most valuable compiler techniques in use today. Independently, each of these optimizations has been shown to result in dramatic improvements. This paper seeks to combine the benefits of both by using a simple memory model to drive optimizations for data locality and parallelism. By unifying the treatment of these optimizations, we are able to place loops with data reuse on inner loops and to introduce parallelism for outer loops. Our strategy produces data locality at the innermost loops, where it is most likely to be exploited by the hardware and places parallelism at the outermost loop, where it is most effective. If these two goals conflict, we present an algorithm that usually reaps the benefits of both.

Optimizing data locality is necessarily both architecture and language dependent. However, the reuse of memory locations and the consecutive access of adjacent memory locations form the foundation of most memory hierarchy optimizations. Reuse of a particular memory reference for arrays can be discovered using

data-dependence analysis [KKP+81]. However, reuse of consecutive accesses, often called *unit stride* access, is a significant source of reuse that can easily be determined when the storage order of arrays and the cache line size is known. In this paper we introduce a simple model for estimating the cost, in memory references, of executing a given loop nest. The principal advantage of this model over previous models is that it takes into account cache reuse due to consecutive accesses to the same cache line. We show how this model can be used to exploit data locality at multiple levels via loop permutation.

Parallelism is usually most effective when it achieves the highest possible *granularity*, the amount of work per parallel task. Granularity is highest when parallel tasks contain the largest amount of work possible. In this paper, parallelism is introduced via the parallel loop construct for shared-memory multiprocessors. Our algorithm first uses the memory model to find a loop organization that exploits data locality. It then seeks to parallelize the outermost loop or a parallel loop that can be positioned outermost. Given sufficient iterations, it then strip mines the loop into two loops, such that one loop is used to achieve locality and the other is used to introduce parallelism.

1.1 Matrix Multiply Example

As an example of this process, consider the ubiquitous matrix multiply.

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

Assuming arrays are stored such that columns of the arrays are in consecutive memory locations, *i.e. column-major order*, this loop organization exploits data locality in the following manner. The consecutive access on the inner I loop to C(I,J) and A(I,K) provide an opportunity for cache line reuse when the cache line size is greater than 1. There is also a loop-invariant reuse of B(K,J) on the I loop. Additionally, the J and the I loops can be parallel. However, if the number of processors, P, is less than the number of iterations of either loop, it is not profitable to utilize both levels of parallelism at once due to additional scheduling overhead. A better execution time would result by maximizing the granularity of one level of the parallelism and then matching

---

it to the machine. If $N = P$, selecting J to be executed in parallel preserves data locality and introduces a single level of parallelism with maximum granularity.

```
    PARALLEL DO J = 1, N
        DO K = 1, N
            DO I = 1, N
                C(I,J) = C(I,J) + A(I,K) * B(K,J)
```

However, if the number of loop iterations is greater than the number of processors, $N > P$, it is often useful to combine independent iterations into a single parallel task to achieve granularity that matches the machine. The parallel loop is strip mined by the number of processors where the strip size is $SS = \lceil N/P \rceil$. We call the J loop the strip and the JJ loop, which walks between strips, the iterator.

```
    PARALLEL DO JJ = 1, N, SS
        DO J = JJ, MIN(JJ + SS - 1, N)
            DO K = 1, N
                DO I = I, N
                    C(I,J) = C(I,J) + A(I,K) * B(K,J)
```

The parallel JJ loop carves up the data space nicely, but if each processor's cache is still not large enough to contain all of array A, *tiling* the loop nest further improves performance by providing reuse of A. Tiling combines strip mining and loop interchange to promote reuse across a loop nest [IT88, Wol89a]. For matrix multiply, the loop nest may be tiled by strip mining the K loop by TS and then interchanging it with J.

```
    PARALLEL DO JJ = 1, N, SS
        DO KK = 1, N, TS
            DO J = JJ, MIN(JJ + SS - 1, N)
                DO K = KK, MIN(KK + B - 1, N)
                    DO I = I, N
                        C(I,J) = C(I,J) + A(I,K) * B(K,J)
```

Here, TS is selected based on the cache size. This organization moves the reuse of A(1:N,KK:KK+TS-1) on the J loop closer together in time, making it more likely to still be in cache.

This optimization approach may be divided into three phases:

1. optimizing to improve data locality,
2. finding and positioning a parallel loop, and
3. performing low-level memory optimizations such as tiling for cache and placing references in registers [LRW91, CCK90].

This paper focuses on the first two phases. We advocate the first two phases be followed by a low-level memory optimizing phase, but do not address it here.

The remainder of this paper is divided into 10 sections. We first present some terms used in this paper to describe data dependence and the machine model. The next section explores and illustrates the effects of parallelism and memory access on performance. The next two sections present a cost model for determining reuse and an algorithm for improving it. Section 6 describes the parallelization strategy. The overall strategy combines the two in Section 7. In Section 8, experimental results are reported. We then overview related work and conclude.

2 Background

2.1 Data Dependence

Dependence analysis is the compile-time analysis of a program's memory accesses. A *data dependence* between two references Ref_1 and Ref_2 indicates that they read or write a common memory location [KKP+81]. True, anti, and output dependences arise when at least one reference is write; the order between Ref_1 and Ref_2 must be preserved to maintain the semantics of the original program. Input dependences arise if both Ref_1 and Ref_2 are reads; they do not restrict program order.

Data dependences may be characterized by their access pattern between loop iterations. The number of loop iterations d separating the source and sink of the dependence is its *dependence distance* [KMC72, Lam74]; it may also be summarized as a *dependence direction* consisting of '$<$', '$=$', or '$>$' [WB87, Wol89b].

Dependence distances and directions are represented as a vector whose elements, displayed left to right, represent the dependence from the outermost to the innermost loop in the nest. By definition all distance and direction vectors are lexicographically positive. We use $\vec{\delta} = (\delta_1, \ldots, \delta_n)$ to represent a distance or direction vector, where δ_i is the dependence distance or direction for the loop at level i.

Dependences may also be characterized as either loop-independent or loop-carried. *Loop-independent* dependences occur on the same iteration of a loop. A dependence between iterations of a loop is called *loop-carried* and prevents the iterations of a loop from being executed in parallel [AK87]. A dependence is carried by the outermost loop for which the element in the direction vector is not an '$=$'.

Data dependence is used to determine the legality of a given loop permutation by checking whether any permuted true, anti, or output dependence vector becomes lexicographically negative [Ban90b, WL90]. Data dependence also characterizes reuse of individual memory locations [CCK90].

2.2 Memory and Language Model

The techniques developed in this paper are intended for shared-memory multiprocessors where each processor has at a local cache and the processors are connected with a common bus. Because we are evaluating reuse, we require some knowledge of the memory hierarchy. However, because our model is very simple, only minimal knowledge of the cache is required; the compiler must know the cache line size (*cls*). The size, set associativity, and replacement policy of the cache are not important here. In addition, we assume a write-back cache and ignore non-unique write references. If the cache is write-through, these writes should be included.

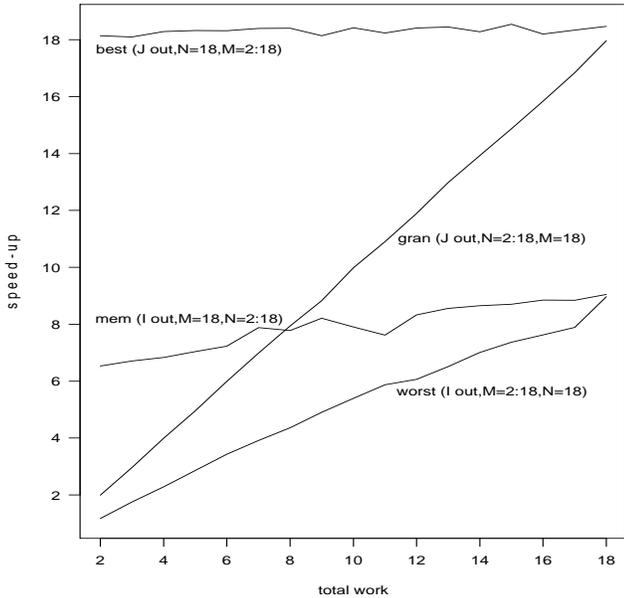In addition, we only concern ourselves with memory

Figure 1: Memory and parallelism tradeoffs

accesses caused by array references, since they dominate memory access in scientific Fortran codes. We also assume that arrays are stored in *column-major* order, where unit stride accesses in the first array dimension translate into contiguous memory accesses. Our results are also valid for *row-major* arrays such as those found in C with only minor changes.

3   Tradeoffs in Optimization

This section illustrates with an experiment the influence of memory reuse and parallelism granularity on speed-up. As expected, it indicates the best performance is possible only when both are utilized effectively in concert. It also shows that when both cannot be achieved at once, there are situations where favoring one or the other results in the best execution time. Neither always dominates. To illustrate, we phrase the following question.

Given enough computation to make parallelism profitable, what is the effect of reuse and how should it affect the optimization strategy?

Figure 1 presents the results of executing different parallel versions of the following loop nest on 18 processors of a Sequent Symmetry S81 with 20 processors, with increasing amounts of total work.

```
DO J = 1, N
    DO I = 1, M
        DO H = 1, L
            C(I, J) = C(I, J) + A(I, J) + B(I, J)
```

The total amount of work is increased by varying the upper bounds N and M from 2 to the number of processors (P = 18). We consider positioning I or J as the

outer parallel loop in the nest. In Figure 1, the *best* version of this loop nest has an outer parallel J loop with 18 iterations (N = 18) and total work is increased by varying M from 2 to 18. Each of the 18 processors accesses distinct columns of each array. This organization exploits cache line reuse on each processor and results in linearly-scalable speed-up.

When the J loop is outermost and the number of parallel iterations of is varied from 2 to 18 along with P and the I loop contains 18 iterations, the total amount of work increases, but the work per processor remains the same. This organization is illustrated by the *gran* line. In this case, the speed-up scales by the number of parallel iterations, but cache line reuse is still facilitated on each processor.

If instead the I loop is made outermost and parallel, then processors must compete for the cache line which contains C(I,J) in order to write it. This competition is called *false sharing*. In addition, multiple processors require cache lines containing A(I,J) and B(I,J), increasing network contention and total memory utilization. When the number of parallel iterations of the I loop as outermost varies from 2 to 18 along with P and the J loop contains 18 iterations, the *worst* line indicates the performance. If the number of parallel iterations of I is held at 18 while the J loop is varied from 2 to 18, the *mem* line results.

Compare the pair of lines *best* and *mem*. The factor of two difference is due to the benefit of cache line reuse in *best*, and the limitations of false sharing and increased bus and memory utilization in *mem*. The same comparison holds for the *gran* and *worst* lines. These results indicate that the parallelizing algorithm must recognize reuse and false sharing to be effective.

Now compare the pair of crossing lines *gran* and *mem*. These computations differ only by an interchange. An optimization strategy that only used loop interchange would be forced to pick between the two. To obtain the best performance for this example, the J loop would be outermost when N > 8, otherwise the I loop should be outermost. In addition, this "crossover" point would need to be determined for each computation, a daunting task. Our approach instead combines loop interchange and strip mining in a parallelization strategy that minimizes false sharing and exploits data reuse.

4   Optimizing Data Locality

In this section we describe two sources of data reuse, then we incorporate both in a simple yet realistic cost model. In subsequent sections, this cost model is used to guide optimizations for improving data locality and exploiting parallelism.

4.1   Sources of Data Reuse

We first consider the two major sources of data reuse.

- multiple accesses to the same memory location
- accesses to consecutive memory locations (*i.e.* stride 1 or unit stride access)

Multiple accesses to the same memory location may arise from either a single array reference or multiple array references. These accesses are loop-independent if they occur in the same loop iteration, and are loop-carried if they occur on different loop iterations. This type of reuse is called *temporal* reuse. The most obvious source of temporal reuse is from loop-invariant references. For instance, consider the reference to A(J) in the following loop nest. It is invariant with respect to the I loop, and is reused by each iteration.

```
DO J = 1,N
    DO I = 1,N
        S = S + A(J) + B(I) + C(J,I)
```

A second source of data reuse is caused by multiple accesses to consecutive memory locations. For instance, each cache line is reused multiple times on the inner I loop for B(I) in the above example. This reuse is called *spatial* reuse. The actual amount of reuse is dependent on the size of B(I) relative to the cache line size and the pattern of intervening references. For the rest of this paper, we assume for simplicity that the cache line size is expressed as a multiple of the number of array elements. For reasonably large computations, references such as C(J,I) do not provide any reuse on the I loop, because the desired cache lines have been flushed by intervening memory accesses.

Previous researchers have studied techniques for improving locality of accesses for registers, cache, and pages [AS79, CCK90, WL91, GJG88]. In this paper we concentrate on improving the locality of accesses for cache; *i.e.* we attempt to increases the locality of access to the same cache line. Empirical results show that improving spatial reuse can be significantly more effective than techniques that consider temporal reuse alone [KMT92]. In addition, consecutive memory access results in reuse at all levels of the memory hierarchy except for registers.

4.2 Simplifying Assumptions

To simplify analysis we make two assumptions. First, our loop cost function assumes that reuse occurs only across iterations of the innermost loop. This assumption decreases precision but greatly simplifies analysis, since it allows the number of cache line accesses to be calculated independent of the permutation of all outer loops. This assumption is accurate if the inner loop contains a sufficiently large number of memory accesses to completely flush the cache after executing all of its iterations. We show later that our optimizations to improve locality can select a desirable permutation of outer loops even with this restriction.

Figure 2: Algorithm RefGroup

INPUT:
  $Refs = \{Ref_1 \ldots Ref_n\}$ references
  $\mathcal{DG} = \{\langle Ref_i \ \vec{\delta} \ Ref_j \rangle, \ldots\}$ the dependence graph
  $l =$ candidate innermost loop
OUTPUT:
  $\{RefGroup_1 \ldots RefGroup_m\}$ reference groups for l
ALGORITHM:
  $m = 0$
  while $Refs \neq \emptyset$ do
    $m = m + 1$
    $RefGroup_m = \{r\}$, where $r \in Refs$
    $Refs = Refs - \{r\}$
    for each $\langle r \ \vec{\delta} \ r' \rangle$ or $\langle r' \ \vec{\delta} \ r \rangle \in \mathcal{DG}$ s.t. $r' \in Refs$
      if $(\delta_l$ is a constant $d)$ & $(\delta_l$ is the only
        nonzero entry in $\vec{\delta}$)
          $RefGroup_m = RefGroup_m + \{r'\}$
          $Refs = Refs - \{r'\}$
      endif
    endfor
  endwhile

Cache interference refers to the situation where two memory locations are mapped to the same cache line, eliminating an opportunity to exploit reuse for one of the references. Our second assumption is that cache interferences occur rarely for small numbers of inner loop iterations, compared to the total number of distinct cache lines accessed in those iterations. In other words, we expect very few interferences for each cache line being reused, since the cache line is only needed for a small number of consecutive inner loop iterations. Lam *et al.* show that this assumption may not hold if cache lines must remain live for longer periods of time. Considerable interference may take place when loops are tiled to increase reuse across outer loops [LRW91].

4.3 Loop Cost Function

Given these assumptions, we present a loop cost function *LoopCost* based on our memory model. Its goal is to estimate the total number of cache lines accessed when a candidate loop l is positioned as the innermost loop. The result is used to guide loop permutation to improve data locality. The estimate is computed in two steps. First, references that will access the same cache line in the same or different iterations of the l loop are combined using *RefGroup*. Second, the number of cache lines accessed by all groups is calculated using *LoopCost*.

4.4 RefGroup

The goal of the *RefGroup* algorithm is to partition variable references in the program text into *reference groups* such that all references in a group access the same memory locations, and consequently the same

cache line. Wolf and Lam call these groups *equivalence classes* exhibiting *group-temporal* reuse. The partition process is particularly simple here because we only consider reuse for each loop when it is positioned innermost.

Two references are in the same reference group for loop l if they actually access some common memory location (data dependence $\vec{\delta}$ exists between them), and the reuse occurs on l if it is positioned as the innermost loop. The common accesses then occur on either the same iteration of l ($\delta_l = 0$) or across d iterations of l ($\delta_l = d$). More formally we define *RefGroup* as follows.

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

1. $\exists\ Ref_1 \vec{\delta} Ref_2$, and

2. $\vec{\delta}$ is a loop-independent dependence, or
   δ_l, the entry in $\vec{\delta}$ corresponding to loop l, is a constant d (d may be zero) and all other entries are zero.

4.4.1 Jacobi Example

For instance, consider the following Jacobi iteration example.

```
DO I = 2,N-1
    DO J = 2,N-1
        A(J,I) = 0.2* (B(J,I) + B(J-1,I) + B(J,I-1)
                    + B(J+1,I) + B(J,I+1))
```

Data dependences connect all references to B. The reference groups for the I loop are:

$\{A(J,I)\}$, $\{B(J,I),B(J,I-1),B(J,I+1)\}$,
$\{B(J-1,I)\}$, $\{B(J+1,I)\}$.

The reference groups for the J loop are:

$\{A(J,I)\}$, $\{B(J,I),B(J-1,I),B(J+1,I)\}$,
$\{B(J,I-1)\}$, $\{B(J,I+1)\}$.

Algorithm *RefGroup* is shown in Figure 2. Its efficiency may be improved by pruning all identical array references, since they access the same memory location on each iteration and always fall in the same reference group.

4.5 LoopCost

After the number of reference groups for loop l is computed with *RefGroup*, the algorithm *RefCost* is applied to estimate the total number of cache lines that would be accessed by each reference group if l were the innermost loop. Once again, the task is simplified because we only consider reuse between iterations of l.

RefCost works by considering one array reference *Ref* from each reference group; these representative references are classified as loop-invariant, consecutive, or non-consecutive with respect to loop l. Loop-invariant array references have subscripts that do not vary with

l; they require only one cache line for all iterations of l.[1] Consecutive array accesses vary with l only in the first subscript dimension. They access a new cache line every *cls* iterations, resulting in *trip/cls* cache line accesses, assuming l performs *trip* iterations. Fewer cache lines are reused for nonunit strides. Nonconsecutive array accesses vary with l in some other subscript dimension; they access a different cache line each iteration, yielding a total of *trip* cache line accesses.

Once *RefCost* is computed, the algorithm *LoopCost* 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. This calculation will underestimate the number of cache lines accessed on the inner loop, if the distance of the dependences for a particular *RefGroup* set are greater than *cls*. Also, slight underestimates occurs because the exact alignment of arrays in memory is not known until run-time. *LoopCost* will overestimate the number of cache lines, if there is additional reuse across an outer loop.

LoopCost is expressed more formally in Figure 3 for the following loop nest containing one array reference from each reference group $RefGroup_1 \ldots RefGroup_m$:

$$do\ i_1 = lb_1, ub_1, s_1$$
$$do\ i_2 = lb_2, ub_2, s_2$$
$$\ldots$$
$$do\ i_n = lb_n, ub_n, s_n$$
$$Ref_1(f_1(i_1,\ldots,i_n),\ldots,f_j(i_1,\ldots,i_n))$$
$$\ldots$$
$$Ref_m(g_1(i_1,\ldots,i_n),\ldots,g_k(i_1,\ldots,i_n))$$

Note that *LoopCost* can be used to calculate cache line accesses even for array references with complex subscript expressions. For instance, it determines that A(I+J+N) results in consecutive memory accesses with respect to both the I and J loops.

4.6 Imperfectly Nested Loops

Because of their simplicity, both *RefGroup* and *LoopCost* can also be applied to imperfectly nested loops. Consider the following example, where the first definition of A(J) is imperfectly nested:

```
DO J = 1, 100
    A(J) = 0
    DO I = 1, 100
        A(J) = A(J) + ...
```

RefGroup would place all references to A(J) in the same reference group. When we apply *RefCost* to calculate the number of cache lines accessed by a reference group, we need to select the most deeply nested member of

---

[1] Of course, loop-invariant references should eventually be put in registers by later optimizations [CCK90].

Figure 3: Algorithm LoopCost

INPUT: $\mathcal{L} = \{l_1, \ldots, l_n\}$ a loop nest with headers lb, ub, s
           $\mathcal{R} = \{Ref_1, \ldots, Ref_m\}$ representatives from each reference group
           $trip_l = (ub_l - lb_l + s_l)/s_l$
           cls = the cache line size,
           $appear(f)$ = the set of index variables that appears in the subscript expression f
           $coeff(i_l, f)$ = the coefficient of the index variable i_l in the subscript f (it may be zero)

OUTPUT:

           $LoopCost(l)$ = number of cache lines accessed with l as innermost loop

ALGORITHM:

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

$\mathbf{RefCost}(Ref_k) =$

1	**if**	$(i_l \notin appear(f_1)) \wedge \ldots \wedge (i_l \notin appear(f_j))$	**loop invariant**
$trip_l/cls$	**if**	$(i_l \in appear(f_1)) \wedge (|coeff(i_l, f_1)| = 1) \wedge (|s_l| = 1) \wedge$	**unit stride**
		$(i_l \notin appear(f_2)) \wedge \ldots \wedge (i_l \notin appear(f_j))$	
$trip_l$	**otherwise**		**no reuse**

the group. *LoopCost* then multiplies the result by the trip counts of all the loops that actually enclose the reference.

5 Loop Permutation

The previous section presents our cost model for evaluating the data locality of a given loop structure with respect to cache. In this section we show how the simplicity and accuracy of the cost model guides loop permutation to restructure a loop nest for better data locality.

A naive optimization algorithm would simply generate all legal loop permutations and select the permutation that yields the best estimated data locality using *LoopCost*. Unfortunately, generating all possible loop permutations takes time that is exponential in the number of loops and can be very expensive in practice. It becomes increasingly unappealing when transformations such as strip mining introduce even larger search spaces.

Instead of testing all possible permutations, we show how our cost model allows us to design an algorithm to directly compute a preferred loop permutation.

5.1 Memory Order

The locality evaluating function *LoopCost* does not calculate data reuse on outer loops; however, we can still restructure programs to exploit outer loop reuse. The key insight is that if loop l causes more reuse than loop l' when both are considered as innermost loops, l will also promote more reuse than l' when both loops are placed at the same outer loop position.

LoopCost can thus be considered to be a measure of the reuse carried by a loop. This allows us to se-

lect a desired permutation of loops called *memory order* that yields the best estimated data locality. We simply rank each loop l using *LoopCost*, ordering the loops from outermost to innermost $(l_1 \ldots l_n)$ such that $LoopCost(l_{i-1}) \geq LoopCost(l_i)$.

5.1.1 Memory Order Algorithm

The algorithm *MemoryOrder* is defined as follows. It computes *LoopCost* for each loop, sorts the loops in order of decreasing cache line accesses (*i.e.* increasing reuse), and returns this loop permutation.

5.1.2 Example

As an example, recall matrix multiply. We compute memory order with $cls = 4$. The reference groups for matrix multiply put the two references to C(I,J) in the same group on all the loops and A(I,K) and B(K,J) are placed in separate groups. *LoopCost* computes the relative reuse on each of the loops as seen below.

references	*LoopCost as innermost*		
	J	K	I
C(I, J)	$n * n^2$	$1 * n^2$	$1/4n * n^2$
A(I, K)	$1 * n^2$	$n * n^2$	$1/4n * n^2$
B(K, J)	$n * n^2$	$1/4n * n^2$	$1 * n^2$
totals	$2n^3 + n^2$	$5/4n^3 + n^2$	$1/2n^3 + n^2$

The algorithm *MemoryOrder* uses these costs to compute a preferred loop ordering of (J, K, I), from outermost to innermost. The same result is obtained by previous researchers [AK84, WL91].

5.2 Permuting to Achieve Memory Order

We must now decide whether the desired memory order is legal. If it is not, we must select some legal loop permutation close to memory order. To determine

whether a loop permutation is legal is straightforward. We permute the entries in the distance or direction vector for every true, anti, and output dependence to reflect the desired loop permutation. The loop permutation is illegal if and only if the first nonzero entry of some vector is negative, indicating that the execution order of a data dependence has been reversed [AK84, Ban90a, Ban90b, WL90].

In many cases, the loop permutation calculated by *MemoryOrder* is legal and we are finished. However, if the desired memory order is prevented by data dependences, we use a simple heuristic for calculating a legal loop permutation near memory order. The algorithm for determining this organization takes $max(D, n^2)$ time in the worst-case where n is the depth of the nest and D is the number of dependences, a definite improvement over considering all legal permutations, which is exponential in n. The algorithm is guaranteed to find a legal permutation with the desired inner loop, if one exists.

5.2.1 Permutation Algorithm

Given a memory ordering $\{i_{\sigma_1}, i_{\sigma_2}, \ldots, i_{\sigma_n}\}$ of the loops $\{i_1, i_2, ..., i_n\}$ where i_{σ_1} has the least reuse and i_{σ_n} has the most, we can test if it is a legal permutation directly by performing the equivalent permutation on the elements of the direction vectors. If the result is a legal set of direction vectors, the loops are permuted accordingly.

Otherwise, we attempt to achieve a "nearby" permutation with the algorithm *NearbyPermutation*. 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. The following theorem holds for the *NearbyPermutation* algorithm.

Theorem: *If there exists a legal permutation where σ_n is the innermost loop, then NearbyPermutation will find a permutation where σ_n is innermost.*

The proof by contradiction of the theorem proceeds as follows. Given an original set of legal direction vectors, each step of the "for" is guaranteed to find a loop which results in a legal direction vector, otherwise the original was not legal [AK84, Ban90a]. In addition, if any loop σ_1 through σ_{n-1} may be legally positioned prior to σ_n it will be.

This characteristic is important because most data reuse occurs on the innermost loop and is due to spatial reuse, so positioning the inner loop correctly will yield the best data locality.

Figure 4: Algorithm NearbyPermutation

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 $\mathcal{O}$

OUTPUT:
$\mathcal{P}$ a nearby permutation of $\mathcal{O}$

ALGORITHM:
$\mathcal{P} = \emptyset$; $k = 0$; $m = n$
while $\mathcal{L} \neq \emptyset$
  for $j = 1, m$
    $l = l_j \in \mathcal{L}$
    if direction vectors for $\{p_1, \ldots, p_k, l\}$ are legal
      $\mathcal{P} = \{p_1, \ldots, p_k, l\}$
      $\mathcal{L} = \mathcal{L} - \{l\}$; $k = k + 1$; $m = m - 1$
      break for
    endif
  endfor
endwhile

6 Parallelism

In the following two subsections, parallelism is evaluated and exploited. We first present a performance estimator that evaluates the potential benefit of parallelism. A parallel code generation strategy then uses performance estimation and the cost model developed in the previous section with other transformations to combine effective parallelism and memory order, making tradeoffs as necessary.

6.1 Performance Estimation

This section uses performance estimation to quantify the effects of parallelism on execution time. Our performance estimator predicts the cost of parallel and sequential performance using a loop model and a *training set* approach.

The goal of our performance estimator is to assist in code generation for both shared and distributed memory multiprocessors [BFKK92, KMM91]. Modeling the target machines at an architectural level would require calculating an analytical model for each supported architecture. Instead our performance estimator uses a training set to characterize each architecture in a machine-independent fashion. A training set is a group of kernel computations that are compiled, executed and timed on each target machine. They measure the cost of operations such as multiplication, branching, intrinsics, and loop overhead. These costs are then made available to the performance estimator via a table of data. Note, the training sets for the performance estimator only measure access times to data in registers or the closest cache.

Of particular interest is the estimation of parallel loops. Given sufficient parallel granularity, using all available processors results in the best execution time.
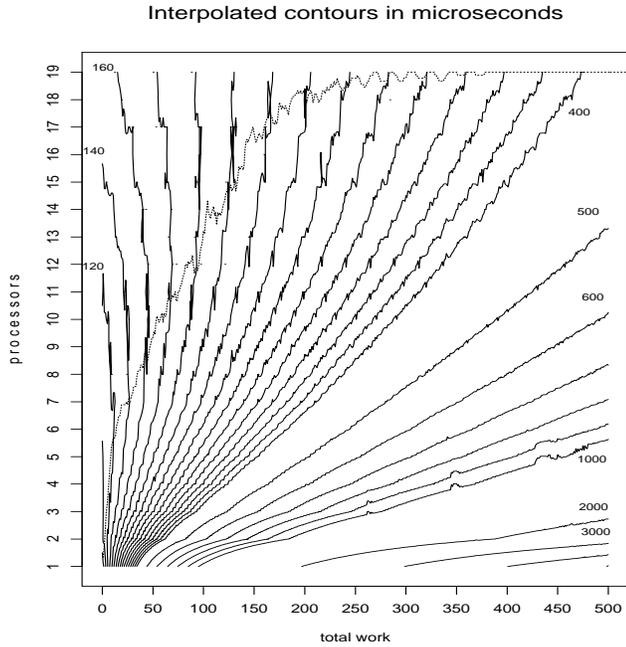
Figure 5: Parallel loop training set

Estimating the cost in this circumstance may be modeled by determining the following.

$$c_s = \text{cost of starting parallel execution}$$
$$c_f = \text{cost of forking and synchronizing}$$
$$\text{a parallel process}$$
$$P = \text{number of processors}$$
$$b = \text{number of iterations of the parallel loop}$$
$$t(B) = \text{cost of the loop body}$$

If the loop bounds are unknown, a guess is used that is based on the declared dimension of the arrays accessed in the loop. With these parameters the performance of a parallel loop with sufficient work may be estimated by:

$$c_s + c_f P + \left\lceil \frac{b}{P} \right\rceil t(B) \ .$$

However, if the amount of work is not sufficient, parallel loop execution is more difficult to model. Instead of an equation, a table is used to indicate the appropriate number of processors for the best performance. The model and the table are generated using a training set.

The sample training set for determining parallel loop overhead begins by varying the total amount of work. For each unit of work, the number of processors is varied from 1 to the total available. The number of processors which minimize the execution time of this work is selected. The result of a training run for parallel loops on the Sequent S81 appears in Figure 5.

This particular training run repeatedly performed a single scalar operation that executed for approximately 10 microseconds, which represents one unit of work in Figure 5. Each of the contour lines indicates a particular execution time. The single line cutting across the

contour lines represents the minimum execution time for executing a particular work load and the appropriate number of processors. When total work is below 250 a table determines the appropriate number of processors and approximate execution time. Once the total work is over about 250, the parallel loop model is used. The estimator provides a single cost function for evaluating loops that chooses between the techniques based on total work and number of loop iterations.

$Estimate(l, how)$ returns $\langle \tau, np \rangle$ where
    l is a loop with body B
    how indicates whether l may be run in parallel

This function returns a tuple $\langle \tau, np \rangle$ with an estimate τ which is the minimal execution time and the number of processors np necessary to obtain the estimate, based on whether the loop is parallel. Note, if the loop is sequential or it is not profitable to run it in parallel, the sequential running time and $np = 1$ are returned.

6.2 Introducing Parallelism

The key to introducing parallelism is to maintain memory order during parallelization by using strip mining and *loop shifting* (loop shifting moves an inner loop outward across one or more loops). Strip mining performs two functions in parallelization. (1) It preserves cache line reuse in parallel execution. Without strip mining, consecutive iterations may be scheduled on different processors, denying cache line reuse. (2) Because strip mining results in two loops, the parallel iterator loop may be shifted outward to maximize granularity while the sequential strip remains in place providing the data locality introduced using memory order. To illustrate this point, consider the subroutine *dmxpy* from Linpackd written in memory order [DBMS79].

```
DO J = JMIN, N2
    DO I = 1, N1
        Y(I) = Y(I) + X(J) * M(I,J)
```

The J loop is not parallel. The I loop can be parallel. Both contain reuse. A simple parallelization that maximizes granularity would interchange the two loops and make the I loop parallel without strip mining. Unfortunately with this organization, the parallel loop may be scheduled such that consecutive iterations are assigned to different processors causing false sharing of Y and eliminating cache line reuse for consecutive accesses to X and M. In addition, cache lines containing the same array elements would be required at multiple processors, increasing total memory and bus utilization.

We instead strip mine a parallel loop by strip size SS $= \lceil N1/P \rceil$ to provide reuse on the strip and parallelize the resultant iterator. If the parallel loop is outermost, as in matrix multiply, parallelization is complete. If not, we use loop shifting to move the parallel iterator to its outermost legal position, maximizing its granularity. Applying this strategy to *dmxpy*, we begin with

the memory ordered loop nest. The I loop is the only parallel loop and it contains reuse. Therefore, it is strip mined. The parallel iterator is not outermost, but it is legally shifted to the outermost position. The compiler shifts the loop, resulting in maximum granularity and data locality as illustrated below.

```
PARALLEL DO I = 1, N1, SS
    DO J = JMIN, N2
        DO II = I, MIN(I + SS - 1, N1)
            Y(II) = Y(II) + X(J) * M(II,J)
```

6.3 Strip Mining Algorithm

If a loop is selected to be performed in parallel, it is strip mined if it contains any reuse. Given sufficient iterations, strip mining exploits data locality and parallelism by using $\lceil N/P \rceil$ as the strip size where N is the number of iterations. Assuming $cls \le P$, the iteration space is sufficiently large if $P < N$. If

$$P < N < cls * P,$$

strip mining by $\lceil N/P \rceil$ is less than the cls and may result in false sharing. However, the granularity of the parallel loop does match P and some reuse will occur. In this case, we still strip mine by $\lceil N/P \rceil$. However, if $N < P$, strip mining may provide reuse but at the cost of drastically reducing the granularity of parallelism. This tradeoff is very machine specific. We choose not to strip mine when $N < P$.

When memory order is computed, the loops are marked to indicate if they contain any reuse. If there is reuse, the strip mining algorithm uses the above equations to select a strip size that maximizes granularity and reuse. If there is no reuse, the strip mining algorithm does not perform strip mining, giving more flexibility to the scheduler.

6.4 Parallelization Algorithm

For memory ordered loop nests that are not parallel on the outermost loop, the *Parallelization* algorithm uses loop shifting to introduce parallelism. It uses loop shifting, rather than a general loop permutation algorithm, in order to minimize the effect of parallelization on data locality. It performs strip mining when the loop contains reuse before shifting for the same reason.

The algorithm for introducing parallelism into memory order appears in Figure 6. It begins by testing whether the outermost loop is parallel. In the first iteration of the "for k" ($j = k = 1$), the first "if" tests if the outermost loop is parallel. Trivially, a shift of loop σ_j to position j is always legal.

If the loop is parallel, it is strip mined and parallelized and the algorithm returns. If the loop is not parallel, a legal shift of an inner loop to position j which is parallel at position j is sought. If a parallel loop is found that can be shifted outermost to j, it is strip mined, parallelized and shifted and the algorithm returns. Otherwise, a shift to position j may cause the

Figure 6: Algorithm Parallelize

INPUT: $\mathcal{L} = \{\sigma_1, \ldots, \sigma_n\}$ a legal permutation

OUTPUT: $\mathcal{T}$ a parallelizing transformation

ALGORITHM:
  $\mathcal{T} = \emptyset$
  for $j = 1, n$
    for $k = j, n$
      if σ_k legal at position j & parallel
        $\mathcal{T} = \{$ **StripMine**$(\sigma_k),$
              shift iterator to j, parallelize it $\}$
        return $\mathcal{T}$
      elseif σ_k legal at j & σ_j becomes parallel
        $\mathcal{T} = \{$**StripMine**$(\sigma_k),$ *shift k iterator to j,*
              StripMine$(new\ \sigma_{j+1}),$
              parallelize the j+1 iterator $\}$
      endif
    endfor
    if $\mathcal{T} \ne \emptyset$ **return** $\mathcal{T}$
  endfor

next inner loop, *i.e.* the loop originally positioned at j, to be parallel. This situation is determined in the "elseif." Because it is more desirable to parallelize a loop at position j than at $j + 1$, all other shifts to position j are considered before this parallelization is returned at the completion of the "for k."

In Figure 6 the *Parallelize* algorithm does not detect when strip mining results in a strip size of less than cls or strip mining is not performed due to insufficient parallel iterations. As we saw in Section 3 these conditions are unavoidable in some cases and the best possible performance is gained even when they hold. However, we extend *Parallelize* as follows to seek a better parallelization for which neither condition holds.

If *StripMine* returns with a strip size of less than cls or does not strip mine due to insufficient parallel iterations, then the number of parallel iterations PI and the size of the strip SS are recorded and the "for k" loop continues instead of returning. If the "for k" finds a parallelization where neither condition holds, it returns. Otherwise, at the completion of the "for k" it selects the parallelization with the largest pair (PI, SS).

7 Optimization Algorithm

The optimization driver for exploiting data reuse and introducing parallelism appears in Figure 7. It combines the component algorithms described in the previous sections and is also O(n^2) time.

It first calls *MemoryOrder* to optimize data locality via loop permutation. It then determines whether the loop contains sufficient computation to pursue parallelism. If it does, the memory ordered loop nest is provided to the algorithm *Parallelize*. If needed, *Parallelize* uses strip mining and loop shifting to introduce loop level parallelism.

Figure 7: Algorithm Optimizer

INPUT: $\mathcal{L} = \{l_1, \ldots, l_n\}$

OUTPUT: $\mathcal{T}$ an optimization of $\mathcal{L}$

ALGORITHM:
  $\mathcal{O} = \mathbf{MemoryOrder}(\mathcal{L})$
  $np = \mathbf{Estimate}\ (\mathcal{O},\ parallel)$
  if $np > 1$ (parallelism is profitable)
    $\mathcal{T} = \mathbf{Parallelize}(\mathcal{O})$
  endif
  perform $\{\ \mathcal{O}, \mathcal{T}\ \}$

The search space in *Parallelize* is constrained to meet our goal of perturbing the memory order as little as possible. If parallelism is not discovered and would be profitable, other optimization strategies that consider all loop permutations, loop skewing [WL90], or loop distribution [McK92] should be explored.

8 Experimental results

We tested the algorithm for optimizing data locality independently and report some of those results here. The overall parallelization strategy was also tested by applying it by hand to several kernels and to the program Erlebacher, provided by Thomas Eidson from ICASE. The results of these experiments are very promising.

8.1 Matrix multiply

We executed all possible loop permutations of matrix multiply for 3 problem sizes, 150×150, 300×300 and 512×512, on a variety of uniprocessors to determine the accuracy of the *MemoryOrder* in predicting the best loop permutations. In Table 1, the permutations are ordered from the most desirable to the least based on the ranking computed by *MemoryOrder*. On most of the processors, memory order JKI produced the best results. On all the processors but the Sequent, the entire ranking generally served to accurately predict relative performance. These results illustrate that *LoopCost* is effective in predicting relative reuse on outer loops as

Table 1: Matrix Multiply (in seconds)

Processor	Loop Permutation					
	JKI	KJI	JIK	IJK	KIJ	IKJ
150×150						
Sequent Weitek	26.0	27.1	31.1	30.7	28.4	26.9
Sun Sparc2	2.33	2.25	3.20	3.16	2.81	2.79
Intel i860	1.16	1.17	1.23	1.18	3.50	3.42
IBM RS6000	0.42	0.46	0.36	0.38	1.08	1.08
300×300						
Sun Sparc2	18.3	17.8	26.1	25.2	24.9	27.1
Intel i860	9.7	10.2	21.7	21.8	59.1	58.9
IBM RS6000	3.37	3.47	12.5	12.5	56.4	56.5
512×512						
Sun Sparc2	91.0	93.6	223	240	277	336
Intel i860	60.2	46.7	143	156	292	292
IBM RS6000	16.7	17.0	183	186	399	399

well as inner loops.

Interestingly, the disparity in execution times between permutations became greater as the processor speed increased. On the individual processors, execution times varied by significant factors of up to 3.69 on the Sparc2, 6.25 on the i860, and a dramatic 23.89 on the RS6000. These results indicate that data locality *should* be the overwhelming force driving scalar compilers today.

Table 2: Speed-ups for Parallel Matrix Multiply

	speed-up of parallel JKI tiled	
	over sequential JKI	over sequential JKI tiled
	19 processors	
150x150	20.5	18.8
300x300	20.1	18.7
	7 processors	
150x150	7.5	6.8
300x300	7.5	7.0

The speed-ups of a parallel tiled matrix multiply on 7 and 19 processors of a Sequent Symmetry S81 for arrays of size 150×150 and 300×300 are presented in Table 2. We ran a sequential version with the loops in memory order JKI, a sequential tiled version, and the identically tiled parallel version. The parallel version is tiled by 4 and is the same version presented in Section 1.1. Besides tiling, no other low-level memory optimizations were used. The speed-ups were basically linear for both matrix sizes when comparing the two tiled versions.

8.2 Dmxpy

The subroutine *dmxpy* from Linpack was optimized using these algorithms as illustrated in Section 6.2. In scientific programs, there are many instances of this type of doubly-nested loop which iterates over vectors and/or matrices, where only one loop is parallel and it is best ordered at the innermost position. These loops may be an artifact of a vectorizable programming style. They appear frequently in the Perfect benchmarks [CKPK90], the Level 2 BLAS [DCHH88], and the Livermore loops [McM86].

Table 3 illustrates the performance benefits with the organization of *dmxpy* generated by our algorithm on matrices of size 200×200 on 19 processors. For comparison, the performance when the I strip is not returned to its best memory position and a parallel inner I loop were also measured.

Table 3: Dmxpy on 19 processors

	loop organization I loop parallel		
	I J II	I II J	J I
speed-up over sequential JI	16.4	13.8	2.9

8.3 Erlebacher

Erlebacher is a tri-diagonal solver for the calculation of variable derivatives written by Thomas Eidson at ICASE, NASA-Langley. It uses 3 dimensional $64 \times 64 \times 64$ arrays. It contains 1341 lines of Fortran. The *Optimizer* algorithm was performed by hand on the entire program. No low-level memory optimizations were performed. The speed-up from this algorithm on 19 processors was 14.2 for the entire application. The speed-up for the parallel portions of the program was 15.0.

9 Related work

Our work bears the most similarity to research by Wolf and Lam [WL91]. They develop an algorithm that estimates all temporal and spatial reuse for a given loop permutation, including reuse on outer loops. This reuse is represented as a *localized vector space*. Vector spaces representing reuse for individual and multiple references are combined to discover all loops $\mathcal{L}$ carrying some reuse. They then exhaustively evaluate all legal loop permutations where some subset of $\mathcal{L}$ is in the innermost position, and select the one with the best estimated locality.

Wolf and Lam's algorithm for selecting a loop permutation is potentially more precise and powerful than the one presented in this paper. It directly calculates reuse across outer loops and can suggest loop skewing and reversal to achieve reuse; however, how often these transformations are needed is yet to be determined. Skewing in particular is undesirable because it reduces spatial reuse.

Gannon *et al.* also formulate the dependence testing problem to give reuse and volumetric information about array references [GJG88]. This information is then used to tile and interchange the loop nests for cache, after which parallelism is inserted at the outermost possible position. They do not consider how the parallelism affects the volumetric information nor if interchange would improve the granularity of parallelism.

Porterfield presents a formula that approximates the number of cache lines accessed, but is restricted to a cache line size of one and loops with uniform dependences [Por89]. Ferrante *et al.* present a more general formula that approximates the number of cache lines and is applicable across a wider range of loops [FST91]. However, they first compute an estimate for every array reference in a loop nest and then combine them, trying not to do dependence testing. Like Wolf and Lam, they exhaustively search for a loop permutation with the lowest estimated cost.

Many algorithms have been proposed in the literature for introducing parallelism into programs. Callahan *et al.* use the metric of minimizing barrier synchronization points via loop distribution, fusion and interchange for introducing parallelism [ACK87, Cal87]. Wolf and Lam [WL90] introduce all possible parallelism via the unimodular transformations: loop interchange, skewing, and reversal. Neither of these techniques try to map the parallelism to a machine, or try take into account data locality, nor is any loop bound information considered. Banerjee also considers introducing parallelism via unimodular transformations, but only for doubly nested loops [Ban90b]. Banerjee does however consider loop bound information.

Because we accept some imprecision, our algorithms are simpler and may be applied to computations that have not been fully characterized in Wolf and Lam's unimodular framework. For instance, we can support imperfectly nested loops, multiple loop nests, and imprecise data dependences. We believe that this approximation is a very reasonable one, especially in view of the fact that we intend to use a scalar cache tiling method as a final step in the code generation process [CCK90]. In addition, the algorithms presented here are $O(n^2)$ time in the worst case, where n is the depth of the loop nest, and are a considerable improvement over work which compares all legal permutations and then picks the best, taking exponential time.

10 Summary and Conclusions

We have addressed the problem of choosing the best loop ordering in a nest of loops for exploiting data locality and for generating parallel code for shared-memory multiprocessors. As our experimental results bear out, the key issue in loop order selection is achieving effective use of the memory hierarchy, especially cache lines. Our approach improves data locality, provides the highest granularity of parallelism, and properly positions loops for low-level memory optimizing transformations. When possible, the benefits of parallelism and data locality are therefore both exploited.

We believe our experimental results provide strong evidence for the effectiveness of this approach. With this method, the programmer is permitted to pay more attention to the correctness of a calculation and less to the explicit loop structure required to achieve high performance.

References

[ACK87]   J. R. Allen, D. Callahan, and K. Kennedy. Automatic decomposition of scientific programs for parallel execution. In *Proceedings of the Fourteenth Annual ACM Symposium on the Principles of Programming Languages*, Munich, Germany, January 1987.

[AK84]   J. R. Allen and K. Kennedy. Automatic loop interchange. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, Montreal, Canada, June 1984.

[AK87]   J. R. Allen and K. Kennedy. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

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

[Ban90a]   U. Banerjee. A theory of loop permutations. In D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*. The MIT Press, 1990.

[Ban90b]   U. Banerjee. Unimodular transformations of double loops. In *Advances in Languages and Compilers for Parallel Computing*, Irvine, CA, August 1990. The MIT Press.

[BFKK92]   V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. A static performance estimator in the Fortran D programming system. In J. Saltz and P. Mehrotra, editors, *Languages, Compilers, and Run-Time Environments for Distributed Memory Machines*. North-Holland, Amsterdam, The Netherlands, 1992.

[Cal87]   D. Callahan. *A Global Approach to Detection of Parallelism*. PhD thesis, Rice University, March 1987.

[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.

[CKPK90]   G. Cybenko, L. Kipp, L. Pointer, and D. Kuck. Supercomputer performance evaluation and the Perfect benchmarks. In *Proceedings of the 1990 ACM International Conference on Supercomputing*, Amsterdam, The Netherlands, June 1990.

[DBMS79]   J. Dongarra, J. Bunch, C. Moler, and G. Stewart. *LINPACK User's Guide*. SIAM Publications, Philadelphia, PA, 1979.

[DCHH88]   J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson. An extended set of Fortran basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 14(1):1–17, March 1988.

[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.

[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.

[KMC72]   D. Kuck, Y. Muraoka, and S. Chen. On the number of operations simultaneously executable in Fortran-like programs and their resulting speedup. *IEEE Transactions on Computers*, C-21(12):1293–1310, December 1972.

[KMM91]   K. Kennedy, N. McIntosh, and K. S. McKinley. Static performance estimation in a parallelizing compiler. Technical Report TR91-174, Dept. of Computer Science, Rice University, December 1991.

[KMT92]   K. Kennedy, K. S. McKinley, and C. Tseng. Improving data locality. Technical Report TR92-179, Dept. of Computer Science, Rice University, March 1992.

[Lam74]   L. Lamport. The parallel execution of DO loops. *Communications of the ACM*, 17(2):83–93, February 1974.

[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, Rice University, April 1992.

[McM86]   F. McMahon. The Livermore Fortran Kernels: A computer test of the numerical performance range. Technical Report UCRL-53745, Lawrence Livermore National Laboratory, 1986.

[Por89]   A. Porterfield. *Software Methods for Improvement of Cache Performance*. PhD thesis, Rice University, May 1989.

[WB87]   M. J. Wolfe and U. Banerjee. Data dependence and its application to parallel processing. *International Journal of Parallel Programming*, 16(2):137–178, April 1987.

[WL90]   M. E. Wolf and M. Lam. Maximizing parallelism via loop transformations. In *Proceedings of the Third Workshop on Languages and Compilers for Parallel Computing*, Irvine, CA, August 1990.

[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.

[Wol89a]   M. J. Wolfe. More iteration space tiling. In *Proceedings of Supercomputing '89*, Reno, NV, November 1989.

[Wol89b]   M. J. Wolfe. *Optimizing Supercompilers for Supercomputers*. The MIT Press, Cambridge, MA, 1989.