Reducing False Sharing and Improving Spatial Locality in a Unified Compilation Framework

Mahmut Kandemir, *Member*, *IEEE*, Alok Choudhary, *Member*, *IEEE Computer Society*, J. Ramanujam, *Member*, *IEEE*, and Prith Banerjee, *Fellow*, *IEEE*

Abstract—The performance of applications on large shared-memory multiprocessors with coherent caches depends on the interaction between the granularity of data sharing, the size of the coherence unit, and the spatial locality exhibited by the applications, in addition to the amount of parallelism in the applications. Large coherence units are helpful in exploiting spatial locality, but worsen the effects of false sharing. A mathematical framework that allows a clean description of the relationship between spatial locality and false sharing is derived in this paper. First, a technique to identify a severe form of multiple-writer false sharing is presented. The importance of the interaction between optimization techniques aimed at enhancing locality and the techniques oriented toward reducing false sharing is then demonstrated. Given the conflicting requirements, a compiler-based approach to this problem holds promise. This paper investigates the use of data transformations in addressing spatial locality and false sharing, and derives an approach that balances the impact of the two. Experimental results demonstrate that such a balanced approach outperforms those approaches that consider only one of these two issues. On an eight-processor SGI/Cray Origin 2000 multiprocessor, our approach brings an additional 9 percent improvement over a powerful locality optimization technique that uses both loop and data transformations. Also, the presented approach obtains an additional 19 percent improvement over an optimization technique that is oriented specifically toward reducing false sharing. This study also reveals that, in addition to reducing synchronization costs and improving the memory subsystem performance, obtaining large granularity parallelism is helpful in balancing the effects of enhancing locality and reducing false sharing, rendering them compatible.

Index Terms—Data reuse, cache locality, false sharing, loop and memory layout transformations, shared-memory multiprocessors.

1 Introduction

Processor speeds have continued to advance at a much higher pace than memory speeds. This has led to the deep memory hierarchies that are found in modern uniprocessor and multiprocessor systems. Although a number of hardware-based techniques are available to exploit these deep memory hierarchies [19], compiler optimizations have come to play an increasingly important role in exploiting the potential performance of these machines. In recent years, there has been a significant amount of work on compiler optimizations aimed at restructuring programs to use the memory hierarchy better, leading to improved performance [51]. In principle, this can be achieved by modifying the access patterns in the program control structures (e.g., loop nests) or by modifying the memory layouts of large data structures (e.g., multidimensional arrays), or through a combination of the two. It has been shown that techniques along these lines are quite successful in enhancing the overall memory performance on uniprocessors [49], [39], [40], [41], [42], [38], [32], [26].

However, there is another critical issue to be considered in the case of shared-memory parallel machines, namely, *false sharing* [8], [46]. False sharing arises when two or more processors that are executing parallel parts of a program access distinct data elements in the same coherence unit [15], [22]. In other words, some form of synchronization is required between the two processors even though there is no data dependence between the computations on the processors. The negative impact of false sharing on the memory performance can be devastating. Eggers and Jeremiassen [15] show that a number of programs—that exhibit good spatial locality on uniprocessors—perform very poorly on multiprocessors. The main reason is the added set of invalidations incurred on updates and the extra invalidation misses that occur when processors reread different data that belong to the invalidated block. Similar observations have been made by others as well [47], [46].

The interaction between locality and false sharing on shared-memory parallel machines is quite well-known. For example, it is known that with smaller cache lines, improving spatial locality also results in a reduction of false sharing. But, at the page level (where the coherence unit is much larger), just targeting spatial locality may not be sufficient [18]. In this paper, we present a mathematical framework that allows us to succinctly represent and study the interaction between the two. More importantly, for array-based regular floating-point scientific codes, this framework enables the derivation of *data transformations* to address the conflicting effects of techniques that improve locality and the techniques that reduce false sharing.

- *M. Kandemir is with the Department of Computer Science and Engineering, The Pennsylvania State University, 220 Pond Laboratory, University Park, PA 16802. E-mail: kandemir@cse.psu.edu.*
- *A. Choudhary and P. Banerjee are with the Department of Electrical and Computer Engineering, Northwestern University, 2145 Sheridan Road, Evanston, IL 60208. E-mail: {choudhar, banerjee}@ece.nwu.edu.*
- *J. Ramanujam is with the Department of Electrical and Computer Engineering, 102 Electrical Engineering Building, S. Campus Drive, Louisiana State University, Baton Rouge, LA 70803-5901. E-mail: jxr@ece.lsu.edu.*

Manuscript received 18 Sept. 2000; revised 1 Nov. 2001; accepted 1 Feb. 2002. For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 112878.

False sharing can be studied at the level of a cache line as well as at the memory page level; our approach does not distinguish between these two types of false sharing. Instead, our interest is in identifying those cases in which optimizations aimed at enhancing locality and optimizations for reducing false-sharing do not conflict with each other. For this purpose, we first represent the potential multiple-writer false sharing in a given loop nest in the form of vectors. Although this representation is approximate, it gives the compiler some idea about the references that may cause a severe form of multiple-writer false sharing if not addressed correctly. We then show how locality optimizations and parallelization techniques affect false sharing. In order to do that, we represent the available locality in a loop nest and the available parallelism options also in mathematical terms. Then, we present an analysis of the cases in which locality optimizations and false sharing optimizations conflict with each other and the cases where they do not. We believe that such an analysis is very useful for compiler writers as well as for the end-users of shared-memory parallel architectures. Our results emphasize the importance of obtaining large-granularity (outermost loop) parallelism. Experimental results on an eight-processor SGI/Cray Origin 2000 machine demonstrate significant improvements. While the importance of outermost loop parallelism has been shown by previous researchers along the lines of reducing synchronization costs [51], reducing interprocessor communication [48], and enhancing memory performance [47], in this paper, we show that obtaining outermost loop parallelism is also important to ensure that enhancing locality and reducing false sharing are not incompatible.

The remainder of this paper is organized as follows: Section 2 presents the relevant background and shows how a specific form of false sharing can be represented in a mathematical framework. Section 3 discusses the impact of loop and data transformations in reducing false sharing and in optimizing locality. In Section 4, we present our heuristic to obtain a balance between enhancing locality and reducing false sharing. Section 5 presents our experimental platform and discusses performance numbers obtained on an eight-processor Origin 2000 distributed-shared-memory multiprocessor. In Section 6, we discuss related work and we conclude in Section 7, with a summary and a brief outline of ongoing and planned research.

2 Preliminaries

Self temporal reuse is said to occur when a reference in a loop nest accesses the same data in different iterations. Similarly, if a reference accesses nearby data, i.e., data residing in the same coherence unit, in different iterations, we say that there is *self spatial reuse* [51]. It should be emphasized that the most useful forms of reuse (temporal or spatial) are those exhibited by the *innermost* loop. If the innermost loop exhibits temporal reuse for a reference, then the accessed element can be placed in a register for the entire duration of the innermost loop (provided that there is no aliasing [51]). Similarly, spatial reuse is most beneficial when it occurs in the innermost loop since, in that case, it may enable unit-stride accesses, leading to repeated accesses to the same coherence unit.

False sharing occurs when two or more processors access (and at least one of them *writes*) different data elements in the same coherence unit (cache line, memory page, etc.) [22], [8], [46], [15]. In this section, we show how to identify a severe form of false sharing that occurs when an array dimension that exhibits *spatial reuse* is accessed by *multiple writers*, i.e., multiple processors that write to data in the same coherence unit. For example, this form of false sharing might occur when each processor updates a different row of a two-dimensional array stored in column-major order. Note that, depending on the array and the size of the coherence units, multiple-writer false sharing can also occur if each processor updates a different column of the said array. However, this type of false sharing occurs only at the boundaries of columns and is not as severe. Now, we introduce two key concepts, namely, the *parallelism vector* and the *reuse summary vector*. Note that, in this paper, we sometimes write a column vector $\bar{x}$ as $(x_1, \ldots, x_n)^T$ when there is no confusion.

The *parallelism vector* indicates which loops in a loop nest *will be* executed in parallel. These loops are a *subset* of the loops that *may be* executed in parallel; this parallelism information is typically obtained through data dependence analysis [51]. Assuming a loop nest of depth n, an element p_i of the parallelism vector $\bar{p} = (p_1, \ldots, p_n)^T$ is one if the iterations of the corresponding loop *will be* executed in parallel, otherwise p_i is zero.

Consider an access to an m-dimensional array in a loop nest of depth n. We assume that the array subscript functions and loop bounds are *affine functions* of enclosing loop indices and symbolic loop-independent parameters [51]. Let $\bar{I}$ denote the *iteration vector* consisting of loop indices starting from the outermost loop to the innermost. Under these assumptions, a reference to an m-dimensional array is represented as $\mathcal{L}\bar{I} + \bar{o}$, where the $m \times n$ matrix $\mathcal{L}$ is called the *access* (or *reference*) *matrix* [49] and the m-element vector $\bar{o}$ is referred to as the *offset* (or *constant*) *vector*. The data reuse theory introduced by Wolf and Lam [49] and later refined by Li [39] is used to identify the types of reuse in a given loop nest. Two iterations represented by vectors $\bar{I}_1$ and $\bar{I}_2$ (where $\bar{I}_1$ precedes $\bar{I}_2$ in sequential execution) access the same data element using the reference represented as $\mathcal{L}\bar{I} + \bar{o}$ if $\mathcal{L}\bar{I}_1 + \bar{o} = \mathcal{L}\bar{I}_2 + \bar{o}$. In this case, the *temporal reuse vector* is defined as $\bar{r} = \bar{I}_2 - \bar{I}_1$, and it can be computed from the relation $\mathcal{L}\bar{r} = \bar{0}$. Assuming *column-major* memory layouts, spatial reuse can occur if the accesses are made to the same column. We can compute the *spatial reuse vector* $\bar{s}$ from the equation $\mathcal{L}_s\bar{s} = \bar{0}$, where $\mathcal{L}_s$ is $\mathcal{L}$ with all elements of the first row replaced by zero [49], [39].

A collection of individual reuse vectors is referred to as a *reuse matrix*. We now focus on spatial reuse vectors. We call the matrix built from these vectors the *spatial reuse matrix*. For a given reuse vector, the first nonzero element from the top (also called the leading element) corresponds to the loop that *carries* the associated reuse. A *reuse summary vector* for a given reuse vector is a vector in which all the elements are zero except the element that corresponds to the loop carrying the reuse (in the associated reuse vector); this element is set to one. The *reuse summary matrix* and the *spatial reuse summary matrix* are defined analogously. Fig. 1 shows an example loop nest and illustrates these concepts. As an example computation, since

$$\mathcal{L}_v = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \end{pmatrix},$$

```
do i = 1, N
 do j = 1, N
  do k = 1, N
   U(i+j,j,i) = V(i,j+k,i+j+1)
  end do
 end do
end do
```

For array U: $\mathcal{L}_u = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$; $\bar{o}_u = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \Rightarrow \bar{r}_u = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$; $\bar{s}_u = \bar{s}'_u = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$

For array V: $\mathcal{L}_v = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \end{pmatrix}$; $\bar{o}_v = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \Rightarrow \bar{s}_v = \begin{pmatrix} 1 \\ -1 \\ 1 \end{pmatrix}$; $\bar{s}'_v = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$

$S = (\bar{s}_u, \bar{s}_v) = \begin{pmatrix} 0 & 1 \\ 0 & -1 \\ 1 & 1 \end{pmatrix}$; $S' = (\bar{s}'_u, \bar{s}'_v) = \begin{pmatrix} 0 & 1 \\ 0 & 0 \\ 1 & 0 \end{pmatrix}$

Fig. 1. An example loop nest and the concepts used in this paper. Notice that the reference to array V does *not* have temporal reuse. $\mathcal{L}_u$ and $\mathcal{L}_v$ are the access matrices and $\bar{o}_u$ and $\bar{o}_v$ are the offset vectors, $\bar{r}_u$ is the temporal reuse vector, $\bar{s}_u$ and $\bar{s}_v$ are the spatial reuse vectors, and $\bar{s}'_u$ and $\bar{s}'_v$ denote the spatial reuse summary vectors. And, S is the spatial reuse matrix and S' is the spatial reuse summary matrix.

the spatial reuse vector should be selected from the null set of

$$\begin{pmatrix} 0 & 1 & 1 \\ 1 & 1 & 0 \end{pmatrix}.$$

Consequently,

$$\bar{s}_v = \begin{pmatrix} 1 \\ -1 \\ 1 \end{pmatrix}.$$

Since the first nonzero element in this vector is the first element, the reuse summary vector is

$$\bar{s}'_v = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}.$$

In this paper, we focus on *self-reuses* (i.e., reuses that originate from individual references to the same array), but our approach can be extended to include *group-reuses* (i.e., reuses that occur when multiple references access the same data element [49]) as well. Unless stated otherwise, all the memory layouts are assumed to be *column-major*.

2.1 Identifying False Sharing Due to an LHS Reference

We begin by noting that a common cause of false sharing is the parallel execution of a loop that carries spatial reuse [39]. For example, in Fig. 2a, parallelizing the i-loop can cause false sharing of array U. The reason is that the spatial reuse for the reference $U(i,j)$ is carried by the i-loop, and parallelizing this loop can cause multiple processors to write to each column of this array. Note that false sharing occurs as a result of the interplay between memory layouts, array subscript functions, coherence unit size, sharing granularity, and parallelization decisions.

We now express the condition for the existence of this form of multiple-writer false sharing in mathematical terms. Let $\bar{s}'$ be a spatial reuse summary vector for a given left hand side (LHS) reference in a nest and let $\bar{p}$ be the parallelism vector for the nest. A severe form of multiple-writer false sharing can occur if $\bar{p}^T\bar{s}' \neq 0$, i.e., if the loop carrying the spatial reuse is parallelized. Considering all the LHS references in the nest, multiple-writer false sharing can occur if $\bar{p}^T S' \neq \bar{0}$, where S' is the spatial reuse summary matrix comprised of the reuse summary vectors for the LHS references. We define the *false sharing vector* $\bar{f}$ as

$$\bar{p}^T S' = \bar{f}^T. \tag{1}$$

The nonzero entries in the false sharing vector $\bar{f}$ identify the references that can cause false sharing. Based on previous work in compilers, one can suggest the desired values for $\bar{p}^T$, S', and $\bar{f}^T$. For example, it is well-known that a desired form of $\bar{p}^T$ has nonzero elements only at the beginning [51]. In effect, most commercial compilers attempt to obtain a single 1 in the leftmost position, which corresponds to parallelizing only the *outermost* loop in the nest. This helps to reduce the synchronization costs [51], reduce interprocessor communication [48], and improve memory performance [47]. On the other hand, previous work on optimizing locality [49], [40], [39] tells us that for each $\bar{s}' \in S'$, the index of the first nonzero element (starting with 1 corresponding to the outermost loop going to n for the innermost loop in an n-nested loop) should be as high as possible. This will ensure that inner loops carry the reuse. In the ideal case, we would prefer the leading element to be the last element in each $\bar{s}'$, i.e., $\bar{s}' = (0, \ldots, 0, 1)^T$. This corresponds to the case where the spatial reuse is carried by the *innermost* loop. In practice, it may not be possible to obtain this ideal reuse summary vector for every reference because of conflicts. Finally, as we have hinted above, the ideal false sharing vector should be a zero vector and the likelihood of multiple-writer false sharing increases with the number of ones in the false sharing vector.

Our focus is on false sharing that is due to one reference per array (self-variable false sharing [11]). It is relatively straightforward to extend the approach presented here to address false sharing due to multiple references to the *same* array. In this case, we need to consider every pair of references to an array that can cause false sharing and, for k such reference pairs, the resulting false sharing vector have k elements. On the other hand, false sharing due to different arrays (multiple-variable false sharing [11]) is, in general, not severe and can be eliminated by the alignment of array variables on coherence unit boundaries; therefore, it is not investigated in this study. Also, we focus mainly on multiple-writer false sharing; reader-writer false sharing can be avoided on machines that employ weak memory

```
do i = 1, N
 do j = 1, N
  do k = 1, N
   U(i,j)=...
   V(i,k)=...
   W(k,j)=...
  end do
 end do
end do
```

(a)

```
do i = i_l, i_u
 do j = j_l, j_u
  do k = k_l, k_u
   U(k,i,j+k)=...
  end do
 end do
end do
```

(b)

```
do i = 1, N
 do j = 1, N
  U(i,j)=...
  V(j,i)=...
  W(i+j,j)= ...
  X(i,i+j)= ...
 end do
end do
```

(c)

```
do i = 1, N
 do j = 1, N
  do k = 1, N
   U(i,j,k)=...
   V(i+j,i+k,j+k)=...
  end do
 end do
end do
```

(d)

```
do i = 2, N-1
 do j = 2, N-1
  U(i,j)=(U(i-1,j)
    +U(i+1,j)
    +U(i,j-1)
    +U(i,j+1))/4.0
 end do
end do
```

(e)

```
do j' = 4, 2N-2
 do i' = max(2,j'-N+1), min(j'-2,N-1)
  U(i',j'-i')=(U(i'-1,j'-i')
    +U(i'+1,j'-i')
    +U(i',j'-1-i')
    +U(i',j'+1-i'))/4.0
 end do
end do
```

(f)

Fig. 2. Several example loop nests that can incur false sharing depending on the parallelization strategy used.

consistency models [11], [19]. However, our approach can be extended to deal with reader-writer false sharing as well.[1]

2.2 Examples

The main issue that we investigate in this paper, is one of simultaneously obtaining large granularity of parallelism, improving spatial locality, and reducing false sharing. We will derive the kind of data transformations useful for achieving this goal. We note that, if we can optimize an LHS reference (which may cause false sharing) such that only the innermost loop carries the spatial reuse for it, then the possibility of multiple-writer false sharing can be reduced if the compiler can derive outermost parallelism *after* the locality optimization. This strategy works fine as long as outermost loop parallelism is available. If this is not the case, then the interplay between locality, parallelism, and false sharing merits further study.

Let us now consider again, the code in Fig. 2a, and show how the false sharing vector is computed. Assume that the nest shown is enclosed by a sequential timing loop and the i-loop is to be parallelized,[2] i.e., $\bar{p} = (1,0,0)^T$. Since

$$\mathcal{L}_u=\begin{pmatrix}1&0&0\\0&1&0\end{pmatrix}, \mathcal{L}_v=\begin{pmatrix}1&0&0\\0&0&1\end{pmatrix}, \text{ and } \mathcal{L}_w=\begin{pmatrix}0&0&1\\0&1&0\end{pmatrix}.$$

The spatial reuse vectors for U, V, and W can be selected from the null sets of $(0,1,0)$, $(0,0,1)$, and $(0,1,0)$, respectively. If we eliminate the candidate vectors that are also temporal reuse vectors (note that we do not consider temporal reuse here), the spatial reuse summary vectors (and also spatial reuse vectors) are $\bar{s}'_u = (1,0,0)^T$, $\bar{s}'_v = (1,0,0)^T$, and $\bar{s}'_w = (0,0,1)^T$; note that we do not consider temporal reuse here. This means that the spatial reuse for arrays U and V are carried by the i-loop and the spatial reuse for W is carried by the k-loop. Thus, the spatial reuse summary matrix is

$$S' = \begin{pmatrix}1&1&0\\0&0&0\\0&0&1\end{pmatrix}.$$

Hence,

$$\bar{f}^T = \bar{p}^T S' = (1,0,0)\begin{pmatrix}1&1&0\\0&0&0\\0&0&1\end{pmatrix} = (1,1,0).$$

This means that, if the i-loop is parallelized, then both the references $U(i,j)$ and $V(i,k)$ may incur multiple-writer false sharing (as they have nonzero entries in the false sharing vector). We also note that, in our example loop nest, if we parallelize the j-loop (instead of the i-loop), then the false sharing vector will be a zero vector (the ideal case), but we will not have outermost loop parallelism anymore. Therefore, there is a trade-off between optimizing for parallelism and reducing false sharing. Ideally, the best parallelism vector is one that enables outermost loop parallelism and maximizes the number of zeroes in the false sharing vector. From $\bar{f}^T = \bar{p}^T S' = \bar{0}$, we obtain $S'^T \bar{p} = \bar{0} \Rightarrow \bar{p} \in Ker\{S'^T\}$. In other words, $Ker\{S'^T\}$ includes all those parallelism vectors that lead to the ideal false sharing vector, namely, the zero vector. From among these candidate parallelism vectors, we need to choose one that is *legal* and that enables the maximum degree of outermost loop parallelism.

We now concentrate on the loop nest shown in Fig. 2b. For the only reference shown, the spatial reuse vector is $\bar{s} = (0,1,-1)^T$, which implies the spatial reuse summary vector $\bar{s}' = (0,1,0)^T$. In order to reduce the extent of false sharing, the parallelism vector $\bar{p}$ should be either $(1,0,0)^T$

1. In fact, multiple-writer false sharing can also be handled by weak memory consistency models. However, it is always better for a compiler to do it since there is cache coherence overhead at runtime.

2. This does *not* mean that the j and k loops cannot be run in parallel, it just means that the compiler decides to parallelize only the outermost loop.

or $(0,0,1)^T$ (assuming only one loop will be parallelized). To exploit outermost loop parallelism, it is better to select $\bar{p} = (1,0,0)^T$, i.e., parallelize the i-loop.

Let us suppose that (for some reason) we want to parallelize the j-loop in this example nest. Further, assume that (i_1, j_1, k_1) and (i_2, j_2, k_2) are two iterations executed in two *different* processors. Assuming column-major memory layouts, multiple-writer false sharing can occur if the following system has a solution:

$$\begin{aligned}
&\textbf{loop bounds condition :}\left\{\begin{array}{c} i_l \le i_1, i_2 \le i_u \\ j_l \le j_1, j_2 \le j_u \\ k_l \le k_1, k_2 \le k_u \end{array}\right\} \\
&\textbf{parallelism condition :}\{j_1 \ne j_2\} \\
&\textbf{stride condition :}\{|k_2 - k_1| < \theta\} \\
&\textbf{locality condition :}\{i_1 = i_2, j_1 + k_1 = j_2 + k_2\},
\end{aligned}$$

where θ is the coherence unit size (specified in number of array elements). The loop bounds conditions ensure that the two iterations are within the loop bounds. The parallelism condition indicates that we parallelize only the j-loop. The locality condition guarantees that the two iterations mentioned will access the same column; therefore, all the subscript positions (dimensions), except maybe the first, should have the same value. And, finally, the stride condition requires that the two accesses fall into the same coherence block (assuming perfect alignment). Assuming that $i_l = j_l = k_l = 1$ and $i_u = j_u = k_u = N$ and $\theta > 1$, the above system has many solutions, e.g., iterations $(1,1,2)$ and $(1,2,1)$. In other words, if we parallelize the j-loop, multiple-writer false sharing is likely to occur.

If, instead, we assume that the i-loop is parallelized, then the parallelism condition will be $i_1 \ne i_2$. Since this condition is in *conflict* with the locality condition (which requires i_1 and i_2 to be equal), the system will not have a solution, meaning that false sharing is unlikely to occur. This formulation of the false sharing problem as a system of equalities and inequalities is very promising (especially with the success of polyhedral algebra tools like the Omega library [30]). However, in this paper, we use a matrix framework and postpone the full treatment of the polyhedral algebraic formulation to a future work. Here, we assume that the loop bounds condition is always satisfied. Also (for portability concerns), we conservatively assume that the stride condition always holds. Thus, what is left to consider are the parallelism and locality conditions.

3 Impact of Transformations on False Sharing

Given a loop nest with a single LHS reference, the compiler's task is to determine suitable values for the vectors $\bar{p}$, $\bar{s}'$, and $\bar{f}$. In order to realize this goal, we consider both loop transformations and data transformations. We first briefly evaluate the effect of loop transformations and, then, make a case for using data transformations.

3.1 Loop Transformations

We assume that the set of applicable loop transformations for an n-deep loop nest are those that can be represented by $n \times n$ nonsingular integer transformation matrix T. From data reuse theory [39], we know that, if $\bar{s}$ is the spatial reuse vector *before* the transformation, then $\bar{s}^+ = T\bar{s}$ is the new spatial reuse vector *after* the transformation. From $\bar{s}^+$, we can easily compute $\bar{s}'^+$, the new spatial reuse summary vector. Unfortunately, finding $\bar{p}^+$, the new parallelism vector *after* the transformation is not as easy. In most cases, we need to run the dependence analyzer to find it. Now, from a loop transformation point of view, we can take three different approaches to the problem.

Parallelism-oriented approach. Using one of the algorithms in the literature (e.g., [50], [40]), we can find a transformation T that results in the best possible parallelism vector $\bar{p}^+$. Then, from $T\bar{s}$, we find the new spatial reuse vector and, finally, using (1), we can check whether the reference incurs false sharing.

Locality-oriented approach. Using one of the algorithms in the literature (e.g., [39], [49]), we can find a transformation T that gives us the best spatial reuse vector $\bar{s}^+$. Then, using dependence analysis, we can find the new parallelism vector and, as before, using (1), we can check whether the reference incurs false sharing.

False sharing-oriented approach. We can try to determine a T that will make the dot-product of $\bar{p}^+$ and $\bar{s}^+$ zero. Unfortunately, it does not seem trivial to find such a loop transformation matrix.[3]

Although we present the alternative strategies here in terms of a reuse summary vector $\bar{s}'$, they can easily be stated using reuse summary matrices by substituting S' for $\bar{s}'$. A problem with loop transformation techniques is that loop transformations impact *both* the parallelism vector and the spatial reuse vector (matrix). That is, in most cases, either some parallelism or some locality must be sacrificed for the sake of the other. What we need is a less intrusive optimization technique such as data transformation, which is explained next.

3.2 Data Transformations

Recently, a number of researchers have proposed data transformations (or, also called *memory layout transformations*) as an alternative to loop transformations for optimizing locality (see [42], [26], [38], [12] and the references therein). In contrast to loop transformations, memory layout transformations are not constrained by data dependences and can be applied to imperfectly-nested loops as well as explicitly-parallelized codes [12]. Moreover, in a given loop nest, the memory layout of each array can be chosen independent of the memory layouts of other arrays. We first summarize our memory layout representation framework presented in [26], [25], and utilized in this work, and then show the effect of data transformations on spatial locality and false sharing.

The working of our framework can be summarized as follows: Each potential layout for an array can be represented using a matrix (explained below). We then construct two different sets of equalities involving these matrices: layout equalities and false sharing equalities (based on false sharing vectors). The objective of the framework is to select a suitable layout (that is, the corresponding matrix) for each array such that both sets of equalities are satisfied as much as possible. This is because we would like to both enhance locality and

3. To see the problem here, assume that $\bar{p}^+ = T\bar{p}$ (see [3] for the cases where this holds). Then, the new false sharing vector is $(T\bar{p})^T TS' = \bar{p}^T T^T TS'$. Since TT^T involves nonlinearity, a trivial solution process is unlikely unless very strict assumptions are made.

eliminate false sharing. In this section, we give the details of this data space oriented optimization strategy.

In our framework, we represent the memory layouts of multidimensional arrays using hyperplanes. In two dimensions, a *hyperplane* defines a set of array elements $(\delta_1, \delta_2)^T$ that satisfy the relation

$$g_1\delta_1 + g_2\delta_2 = c \tag{2}$$

for some constant c. In this equation, g_1 and g_2 are rational numbers called *hyperplane coefficients* and c is a rational number referred to as the *hyperplane constant* [20], [44]. The hyperplane coefficients in (2) can be written as a hyperplane vector $\bar{g} = (g_1, g_2)^T$. A *hyperplane family* is a set of hyperplanes defined by the same $\bar{g}$ and different values of c.

A hyperplane family can be used to partially define the memory layout of a multidimensional array [25]. In a two-dimensional data (array) space, a hyperplane family defines parallel hyperplanes (lines), each corresponding to a different value of c. We assume that the array elements on a specific hyperplane are stored in consecutive memory locations. As an example, for an array whose memory layout is column-major, each column represents a hyperplane (a line) whose elements are stored in consecutive locations in memory. Given a large array, the relative storage order of the columns (with respect to each other) is not important to us in this paper. Therefore, we represent the column-major layout with the hyperplane vector $\bar{g} = (0, 1)^T$, which simply indicates the orientation of the hyperplanes. Similarly, the vectors $(1, 0)^T$, $(1, -1)^T$, and $(1, 1)^T$ correspond to row-major, diagonal, and antidiagonal memory layouts, respectively. Two array elements $\bar{\delta} = (\delta_1, \delta_2)^T$ and $\bar{\delta}' = (\delta_1', \delta_2')^T$ belong to the same hyperplane $\bar{g} = (g_1, g_2)^T$ if and only if

$$(g_1, g_2)(\delta_1, \delta_2)^T = (g_1, g_2)(\delta_1', \delta_2')^T. \tag{3}$$

As an application of (3), consider an array stored in column-major order, i.e., the layout hyperplane vector is $(0, 1)^T$. Here, the array elements $(2, 3)^T$ and $(5, 3)^T$ belong to the same hyperplane (i.e., same column), whereas the elements $(2, 3)^T$ and $(2, 4)^T$ do not. We say that two array elements that belong to the same hyperplane have *spatial locality* [25]. Although this definition of spatial locality is somewhat coarse (e.g., does not hold at the array boundaries) and is different from the definitions used in previous work [49], [39], it is sufficient for the purposes of this paper.

In a two-dimensional space, a single hyperplane family is sufficient to partially define a memory layout. In higher dimensions however, we may need to use more hyperplane families. Let us concentrate on a three-dimensional array U whose layout is column-major. Such a layout can be represented using two hyperplanes: $\bar{g} = (0, 0, 1)^T$ and $\bar{g}' = (0, 1, 0)^T$. We can write these two hyperplanes collectively as a *layout constraint* matrix or simply a *layout* matrix

$$G_u = \begin{pmatrix} \bar{g}^T \\ \bar{g}'^T \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}.$$

In that case, two data elements $\bar{\delta}$ and $\bar{\delta}'$ have spatial locality if *both* of the following conditions are satisfied: $\bar{g}^T\bar{\delta} = \bar{g}^T\bar{\delta}'$ and $\bar{g}'^T\bar{\delta} = \bar{g}'^T\bar{\delta}'$. The elements that have spatial locality should be stored in consecutive memory locations. Note how this layout representation matches the column-major layout of a three-dimensional array in Fortran. For such an array, in order for two elements to have spatial locality (according to our definition), all the array indices, except maybe the first one, should be equal. Notice that the two conditions given above ensure that these index equalities hold. This representation framework can easily accommodate higher-dimensional arrays. We refer the interested reader to [26] and [25] for an indepth discussion of hyperplane-based layout representations.

It is important to note that memory layout transformations do *not* have any effect on the parallelism vector. This is an important advantage over loop transformations. As a result, *we can start with the best possible parallelization strategy and then use data transformations to strike a balance between spatial locality and false sharing without disturbing the available parallelism*. This is the approach taken in this paper. Let $\bar{I}$ and $\bar{I}'$ be two iteration vectors and let $\bar{s}$ denote $\bar{I}' - \bar{I}$. The data elements accessed by these two vectors through a reference represented by $\mathcal{L}$ and $\bar{o}$ to a two-dimensional array are $\mathcal{L}\bar{I} + \bar{o}$ and $\mathcal{L}\bar{I}' + \bar{o}$, respectively. Using (3) given above, these two elements have spatial locality if (where $\bar{s}$ denotes the spatial reuse vector)

$$\bar{g}^T(\mathcal{L}\bar{I} + \bar{o}) = \bar{g}^T(\mathcal{L}\bar{I}' + \bar{o}) \Rightarrow \bar{g}^T\mathcal{L}(\bar{I}' - \bar{I}) = 0 \text{ or } \bar{g}^T\mathcal{L}\bar{s} = 0. \tag{4}$$

Now, we have two important equations, (1) and (4), both related to locality. The former gives the relationship between parallelization decisions and locality, whereas the latter shows the relationship between locality and memory layout. Let us concentrate on a single LHS reference with one spatial reuse vector in an n-deep loop nest. Assume that we want to parallelize only the outermost loop in the nest, i.e., $\bar{p} = (1, 0, \ldots, 0)^T$. In order to reduce the chances for multiple-writer false sharing due to this reference, $\bar{p}^T\bar{s}'$ should be zero. Substituting the value for $\bar{p}$, we get $(1, 0, \ldots, 0)\bar{s}' = 0 \Rightarrow \bar{s}' = (0, \times, \ldots, \times, \times)^T$, where $\times$ stands for *dont-care*.[4] A simple spatial reuse vector $\bar{s}$ that satisfies the $\bar{s}'$ vector above is $(0, \ldots, 0, 1)^T$. If we substitute this spatial reuse vector in (4), we can find an appropriate memory layout using $\bar{g}^T \in Ker\{\mathcal{L}\bar{s}\}$, or $\bar{g}^T \in Ker\{\bar{l}_n\}$, where $\bar{l}_n$ is the last column of $\mathcal{L}$.

What we have done here is (assuming outermost loop parallelism) to find a spatial reuse vector and, then, by using that vector to find a memory layout. Notice that the spatial reuse vector that we derived reduces false sharing. It is important to note that such a vector is *ideal* from the spatial locality point of view as well. Li [39] has observed that the form of the ideal spatial reuse vector is $(0, \ldots, 0, 1)^T$ since it exploits spatial locality in the innermost loop. To sum up, in this case, we are able to reduce false sharing and optimize spatial locality together. In general (after obtaining maximum granularity parallelism using loop transformations), from a data transformation point of view, we can define the problem as one of *finding a memory layout* such that

1. false sharing will be reduced and
2. spatial locality will be enhanced.

Once we determine a suitable layout, it is relatively easy to implement it (see [38], [42]) in a compiler that assumes a

4. $\times$ can be 0 or 1; there can be only a single 1 according to our definition of a *reuse summary vector*.

default memory layout, e.g., column-major layout in Fortran. This is achieved by transforming the default layout matrix (e.g., that represents column-major layout) to the desired layout matrix. The linear transformation matrix that achieves this is then used to transform the subscript expressions (of array references) as well as array declarations. Determining the transformation matrix and transforming subscript expressions are relatively easy; they are similar to the code modifications caused by loop transformations. Modifying array declarations is in general more difficult as such modifications might increase the overall data space requirements of the code. These extra data space requirements can be reduced significantly using the techniques discussed in Leung and Zahorjan [38].

Let us now consider the loop nest shown in Fig. 2c. Assuming that the outermost loop i is parallelized, from $(1,0)\bar{s}' = 0$, we obtain $\bar{s}' = (0,\times)^T$. Using this summary vector, we can select $\bar{s} = (0,1)^T$ for all the references in the nest.

$$\text{For array } U: \quad \bar{g}^T \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = 0 \quad \Rightarrow$$

$$\bar{g}^T \in Ker\left\{ \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right\} \quad \Rightarrow \quad \bar{g} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

$$\text{For array } V: \quad \bar{g}^T \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = 0 \quad \Rightarrow$$

$$\bar{g}^T \in Ker\left\{ \begin{pmatrix} 1 \\ 0 \end{pmatrix} \right\} \quad \Rightarrow \quad \bar{g} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

$$\text{For array } W: \quad \bar{g}^T \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = 0 \quad \Rightarrow$$

$$\bar{g}^T \in Ker\left\{ \begin{pmatrix} 1 \\ 1 \end{pmatrix} \right\} \quad \Rightarrow \quad \bar{g} = \begin{pmatrix} 1 \\ -1 \end{pmatrix}$$

$$\text{For array } X: \quad \bar{g}^T \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = 0 \quad \Rightarrow$$

$$\bar{g}^T \in Ker\left\{ \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right\} \quad \Rightarrow \quad \bar{g} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}.$$

With these hyperplane vectors, it is clear that the arrays U and X should be row-major, array V should be column-major, and array W should have a diagonal memory layout (see the discussion in Section 3.2). Note that (provided the i-loop is parallel) these layouts do not incur severe multiple-writer false sharing and lead to good spatial locality in the innermost j-loop.

An important question now is, under what circumstances we *cannot* optimize spatial locality and reduce false sharing without any conflict. Before answering this question, consider the loop nest shown in Fig. 2d. Assume that we parallelize both the i and j loops (provided it is legal to do so). Thus, in mathematical terms, $\bar{p} = (1,1,0)^T$. Such a two-level parallelization might be useful in architectures like Convex Exemplar where there are two levels of processor hierarchies (interhypernode and intrahypernode). From $(1,1,0)\bar{s}' = 0$, we obtain $\bar{s}' = (0,0,\times)^T$. Using this summary vector, we can select $\bar{s} = (0,0,1)^T$ for both the references in the nest. Therefore,

$$\text{For array } U: \quad \bar{g}^T \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} = 0 \quad \Rightarrow$$

$$\bar{g}^T \in Ker\left\{ \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \right\} \quad \Rightarrow \quad G_u = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

$$\text{For array } V: \quad \bar{g}^T \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} = 0 \quad \Rightarrow$$

$$\bar{g}^T \in Ker\left\{ \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix} \right\} \quad \Rightarrow \quad G_v = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & -1 \end{pmatrix}.$$

As in the previous example, with these layouts,[5] we are able to reduce false sharing and optimize spatial locality together. In fact, it is easy to see that a parallelism vector such as $(1,\ldots,1,0,\ldots,0)^T$ can always be treated as $(1,0,\ldots,0)^T$; that is, all the outermost parallel loops can be collapsed into one loop. Thus, we can conclude the following:

> In a given parallelism vector, if all the ones are in the leftmost positions consecutively (without a zero in between them), then it is possible to reduce false sharing and optimize locality together for a given left-hand side reference.

It is also useful to verify the solution proposed for the example in Fig. 2d, using a system of constraints. Let us focus on just array V. The following system should have at least two iterations (i_1, j_1, k_1) and (i_2, j_2, k_2) as a solution in order for multiple-writer false sharing to occur through the reference $V(i+j, i+k, j+k)$ in the original loop nest.

$$\textbf{loop bounds condition :}\left\{ \begin{array}{c} 1 \le i_1, i_2 \le N \\ 1 \le j_1, j_2 \le N \\ 1 \le k_1, k_2 \le N \end{array} \right\}$$

$$\textbf{parallelism condition :}\left\{ \begin{array}{c} i_1 \ne i_2 \\ j_1 \ne j_2 \end{array} \right\}$$

$$\textbf{stride condition :}\{|(i_2 + j_2) - (i_1 + j_1)| < \theta\}$$

$$\textbf{locality condition :}\{i_1 + k_1 = i_2 + k_2, j_1 + k_1 = j_2 + k_2\},$$

where θ is the coherence unit size in array elements. It is easy to see that this system has many solutions. On the other hand, if we apply our layout transformations, the new reference will be $V(i+k, i+j, i-j)$.[6] Now, the layout condition becomes $\{i_1 + j_1 = i_2 + j_2$ and $i_1 - j_1 = i_2 - j_2\}$, which implies $i_1 = i_2$. This, in turn, conflicts with the parallelism condition $i_1 \ne i_2$. That is, the new system has no solution meaning that the possibility of multiple-writer false sharing is reduced.

3.3 Outer and Inner Loop Parallelization

It may not always be possible to obtain outermost loop parallelism in loop nests [36]. For an n-nested loop, let D denote the dependence distance matrix [50], the columns of

5. Here, G_u here, corresponds to a row-major layout and G_v represents a nonconventional layout. See [25] and [26] for a precise interpretation of layout matrices for three and higher dimensional arrays.

6. See [38], [42], [26] for details of rewriting access matrices and array declarations after layout transformations.

which are the (constant) dependence distance vectors [51]. If $rank(D) < n$, then the outermost $n - rank(D)$ loops can be run in parallel. If $rank(D) = n$, then the loop nest can be transformed such that the outermost is *sequential*, but the inner $n-1$ loops can be run in parallel [36]. As noted earlier in this paper, in the case of an outermost parallel loop, we set the parallelism vector to $(1,0,\ldots,0,0)^T$ and then optimize each reference using $(0,0,\ldots,0,1)^T$ as the spatial reuse vector. *This allows us to optimize spatial locality and reduce false sharing together*. If $rank(D) = n$, then outermost loop parallelism is *not* available and, therefore, optimizing for improving spatial locality and optimizations for reducing false sharing will conflict.

Consider now, the loop nest shown in Fig. 2e. This nest represents the core computation in the successive-over-relaxation (SOR) code. Both the i and the j loops carry data dependences, so, as it is, none of the loops can be executed in parallel. By skewing the inner loop with respect to the outer loop, followed by interchanging the loops, we derive the code shown in Fig. 2f. Now, the innermost loop (i') can run in parallel, giving a parallelism vector $(0,1)^T$. In order to reduce false sharing, we need to choose $\bar{s} = (1,0)^T$. Using this reuse vector,

$$\bar{g}^T \begin{pmatrix} 0 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = 0,$$

i.e., $\bar{g}^T \in Ker\{(0,1)^T\}$, or $\bar{g} = (1,0)^T$. Thus, the layout of the array should be row-major. With this choice, there is no false sharing due to multiple writes to the LHS reference, but spatial locality is very poor as successive iterations of the local portion of a processor touch different rows of the array.

Let us now find the result of using a locality-oriented approach for the same nest. We use $\bar{s} = (0,1)^T$ as our (best) spatial reuse vector. From

$$\bar{g}^T \begin{pmatrix} 0 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = 0,$$

we get $\bar{g}^T \in Ker\{(1,-1)^T\}$, or $\bar{g} = (1,1)^T$. Here, the layout of the array should be antidiagonal. Now, we have spatial locality exploited in the innermost loop since successive iterations of the innermost loop access a given antidiagonal, but we incur false sharing at the coherence block boundaries. This example clearly shows the potential conflict between optimizing spatial locality and reducing false sharing.

4 Heuristic for Reducing False Sharing and Enhancing Spatial Locality

In this section, we propose a solution for enhancing spatial locality and reducing false sharing together. Note that, while false sharing is an issue for arrays that have at least one reference on the LHS, spatial locality is an issue for all the arrays referenced in the nest. Therefore, we divide the arrays referenced in the nest being analyzed into two groups: 1) arrays referenced on the RHS only and 2) arrays referenced on both sides. Supposing that there is a total of γ arrays referenced in the nest, $A_1, A_2, \ldots, A_\delta, A_{(\delta+1)}, \ldots, A_{(\gamma-1)}, A_\gamma$. Without loss of generality, we can assume that δ of these arrays fall into the first group and $\gamma - \delta$ fall into the second. In the following, we do *not* distinguish between spatial reuse vector and spatial reuse summary vector, as these two vectors are usually the same for most loop nests that we come across in practice.

The first group is easy to handle. Since false sharing is not an issue for this group, all we need is to use (4) for optimizing locality. Specifically, let us consider an array j where $1 \le j \le \delta$. Assume that the number of references to this array is t_j. We can use the constraint $\bar{g}_j^T \mathcal{L}_{jk} \bar{s}_{jk} = 0$ to find the optimal layout for this array where $\mathcal{L}_{jk}$ is the kth reference to this array and $\bar{s}_{jk}$ is the kth spatial reuse vector. In the ideal case, we want to choose $\bar{s}_{jk} = (0,\ldots,0,1)^T$ for each reference k $(1 \le k \le t_j)$. However, given a large number of references, this may not be possible. That is, different references to the same array may impose conflicting layout requirements. In that case, using profile information, we favor some references over the others and optimize for only those favored references. In the following, we briefly discuss a profile-based reference selection scheme; see [25] for an alternative method. For each reference $\mathcal{L}_{jk}$, we associate a weight function *weight* ($\mathcal{L}_{jk}$), which gives the number of times this reference is touched in a typical execution of the program at hand. We use profiling to get the values of *weight* ($\mathcal{L}_{jk}$). Then, the solution process is as follows:

1. Set $\bar{s}_{jk} = (0,\ldots,0,1)^T$ for each reference k $(1 \le k \le t_j)$.
2. Sort the reference according to their *weights* in nonincreasing order.
3. Attempt to solve $\bar{g}_j^T \mathcal{L}_{jk} \bar{s}_{jk} = 0$ for each k.
4. If there is a solution, return; else omit the reference with the smallest weight, and go to Step 3.

When the process terminates, we will have $\eta_j \le t_j$ references optimized for the locality in the *innermost* loop. This process is independently repeated for all δ arrays in the first group $(1 \le j \le \delta)$.

As for the second group, false sharing might be an important issue especially at the page-level. Let us now focus again on a single array j where $\delta + 1 \le j \le \gamma$. We divide the references for this array into two groups:

1. the LHS references, $\mathcal{L}_{jk}$ where $1 \le k \le \Delta$ and
2. the RHS references, $\mathcal{L}_{jk'}$ where $\Delta + 1 \le k' \le t_j$.

The constraints to be satisfied for each group are different and can be listed as follows:

for the LHS references		for the RHS references
$\bar{p}^T \bar{s}_{j1} = 0$	$\bar{g}_j^T \mathcal{L}_{j1} \bar{s}_{j1} = 0$	$\bar{g}_j^T \mathcal{L}_{j(\Delta+1)} \bar{s}_{j(\Delta+1)} = 0$
$\bar{p}^T \bar{s}_{j2} = 0$	$\bar{g}_j^T \mathcal{L}_{j2} \bar{s}_{j2} = 0$	$\bar{g}_j^T \mathcal{L}_{j(\Delta+2)} \bar{s}_{j(\Delta+2)} = 0$
$\vdots$	$\vdots$	$\vdots$
$\bar{p}^T \bar{s}_{j\Delta} = 0$	$\bar{g}_j^T \mathcal{L}_{j\Delta} \bar{s}_{j\Delta} = 0$	$\bar{g}_j^T \mathcal{L}_{jt_j} \bar{s}_{jt_j} = 0$

In this case, we try the following two options and select the one that performs better.

- In the first option, we set:

$$\bar{s}_{j1} = \bar{s}_{j2} = \ldots = \bar{s}_{j\Delta} \in Ker\{\bar{p}^T\} \quad \text{and}$$
$$\bar{s}_{j(\Delta+1)} = \bar{s}_{j(\Delta+2)} = \bar{s}_{jt_j} = (0,\ldots,0,1)^T.$$

  In other words, in this option, for the LHS references, we favor reducing false sharing over enhancing locality and, for the RHS references, we are trying to maximize locality. After these settings, we attempt to solve the constraints given above for

```
INPUT: A loop nest that accesses the arrays $A_1, A_2, ..., A_\delta, A_{\delta+1}, ...., A_\gamma$
OUTPUT: An optimized loop nest with layout-transformed arrays
Begin
  Using a parallelization algorithm obtain largest granular parallelism; i.e., determine $\bar{p}^T$
  Let $\mathcal{A} = \{A_1, A_2, ..., A_\delta\}$ be the arrays that do not have any LHS reference
  Let $\mathcal{B} = \{A_{\delta+1}, A_{\delta+2}, ...., A_\gamma\}$ be the remaining arrays
  Foreach $A_j \in \mathcal{A}$ do
    Order the references according to their dynamic occurrences (weights)
    Let $\mathcal{L}_{j1}, \mathcal{L}_{j2}, ..., \mathcal{L}_{jt_j}$ be the (ordered) access matrices for the references to this array
    Set $\bar{s}_{jk} = (0, ..., 0, 1)^T$
    Set solution = false
    While (not solution)
      Solve the system $\bar{g}_j^T \mathcal{L}_{jk} \bar{s}_{jk} = 0$ for $\bar{g}_j^T$
      If there is a solution, then set solution = true;
        else omit the constraint of the reference with the smallest weight
    EndWhile
  EndForeach
  Foreach $A_j \in \mathcal{B}$ do
    Order the references according to their dynamic occurrences (weights)
    Let $\mathcal{L}_{j1}, \mathcal{L}_{j2}, ..., \mathcal{L}_{j\Delta}$ be the (ordered) access matrices for the LHS references to this array
    Let $\mathcal{L}_{j(\Delta+1)}, \mathcal{L}_{j(\Delta+2)}, ..., \mathcal{L}_{jt_j}$ be the (ordered) access matrices for the RHS references to this array
    //* Option 1 *//
    Set $\bar{s}_{jk} \in Ker\{\bar{p}^T\}$ for $1 \le k \le \Delta$
    Set $\bar{s}_{jk'} = (0, ..., 0, 1)^T$ for $\Delta + 1 \le k' \le t_j$
    Set solution = false
    While (not solution)
      Solve the system $\bar{p}^T \bar{s}_{jk} = 0$; $\bar{g}_j^T \mathcal{L}_{jk} \bar{s}_{jk} = 0$; $\bar{g}_j^T \mathcal{L}_{jk'} \bar{s}_{jk'} = 0$
      If there is a solution, then set solution = true;
        else omit the constraint of the reference with the smallest weight
    EndWhile
    Compute cumulative weight(Option 1) as the sum of the weights of the references that are satisfied
    //* Option 2 *//
    Set $\bar{s}_{jk} = (0, ..., 0, 1)^T$ for $1 \le k \le \Delta$
    Set $\bar{s}_{jk'} = (0, ..., 0, 1)^T$ for $\Delta + 1 \le k' \le t_j$
    Set solution = false
    While (not solution)
      Solve the system $\bar{p}^T \bar{s}_{jk} = 0$; $\bar{g}_j^T \mathcal{L}_{jk} \bar{s}_{jk} = 0$; $\bar{g}_j^T \mathcal{L}_{jk'} \bar{s}_{jk'} = 0$
      If there is a solution, then set solution = true;
        else omit the constraint of the reference with the smallest weight
    EndWhile
    Compute cumulative weight(Option 2) as the sum of the weights of the references that are satisfied
    Compare cumulative weight(Option 1) and cumulative weight(Option 2)
    Select the option with the larger weight from the previous step
  EndForeach
  Using $\bar{g}_j^T$ vectors found ($1 \le j \le \gamma$) layout-transform the arrays in the nest
End
```

Fig. 3. A balanced algorithm that reduces false sharing while improving locality.

$\bar{g}_j$. As before, if there is no solution, we omit the (constraints belonging to the) reference with the smallest *weight* and try to solve the system again.

- In the second option, we set:

$$\bar{s}_{j1} = \bar{s}_{j2} = \ldots = \bar{s}_{j\Delta} = \bar{s}_{j(\Delta+1)} = \bar{s}_{j(\Delta+2)} = \bar{s}_{jt_j} = (0, \ldots, 0, 1)^T.$$

That is, we favor optimizing locality over reducing false sharing for both LHS and RHS references. The rest of the process is the same as in the previous option.

We compare the solutions from these two options and select the best one. Our comparison scheme is rather simple. For each option, we calculate a *cumulative weight*, which is the sum of the weights of the references that are satisfied (i.e., *not omitted* during the solution process). We prefer the option with the larger cumulative weight. The overall algorithm is given in Fig. 3.

Note that, in general, a layout that is suitable for one array in one loop nest may not be suitable for the same array in another loop nest. This means that, in contrast to loop-based transformation techniques that can handle a single loop nest at a time, an optimization strategy based on layout transformations should consider all the nested loops that access the array whose layout is to be manipulated. Our

```
do i = 1, N
   do j = 1, N
      U(i,j)   = V(j,i-1)+2.0
      ...      = ...
      V(j,i)   = W(i+j,i)+1.0
   end do
end do

do i = 1, N
   do j = 1, N
      do k = 1, N
         X(i+k,j+k)  = ...
         ...         = X(i+j,k-j-N)
         ...         = ...
      end do
   end do
end do
```

(a)

```
dopar j = 1, N
   do i = 1, N
      U(i,j)   = V(j,i-1)+2.0
      ...      = ...
      V(j,i)   = W(i+j,i)+1.0
   end do
end dopar

do i = 1, N
   do j = 1, N
      dopar k = 1, N
         X(i+k,j+k)  = ...
         ...         = X(i+j,k-j-N)
         ...         = ...
      end dopar
   end do
end do
```

(b)

Fig. 4. (a) An example program fragment. (b) Parallelism-optimized version of (a) with parallel loops identified using `dopar`.

solution to this global optimization problem is as follows: First, we determine an order of processing the nests; that is, if a nest is more important (costly) than another, we optimize the more important nest first. Again, profiling is used to determine the estimated cost of a nest, which can be defined as the sum of the weights (number of runtime occurrences) of the references it encloses. Then, for the most important nest, we optimize it using the approach explained in this paper. After optimizing this nest, the memory layouts of some of the arrays referenced in it will be fixed. Then, we consider the next important nest, and optimize it using a slightly different version of our approach that takes the layouts found in the most important nest into account.[7] Then, we move to the third most important nest and, in optimizing it, we take all the layouts determined so far (in the most important and the second most important nests) into account, and so on. The details of the global layout propagation algorithm is outside the scope of this paper; it is similar to those presented in [28], [27]. It should also be noted that a more sophisticated optimization strategy could use loop transformations (in addition to data transformations) during the optimization of the nests for false sharing.

4.1 Example Application of the Algorithm

We now present an example application of our heuristic. Consider the program fragment given in Fig. 4a. Let us first focus on the first loop nest. Our approach starts with optimizing parallelism. In this nest, we can achieve a maximum granular parallelism by interchanging the loops i and j. The resulting nest is shown in Fig. 4b. The access matrices for the arrays referenced in this nest in Fig. 4b are

$$\mathcal{L}_{u1} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \mathcal{L}_{v1} = \mathcal{L}_{v2} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \text{ and } \mathcal{L}_{w1} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}.$$

We next parallelize the outer i-loop; that is, $\bar{p}^T = (1, 0)$. The array W belongs to the first group mentioned above, i.e., it has no reference on the LHS. Therefore, the only constraint that should be satisfied is $\bar{g}_w^T \mathcal{L}_{w1} \bar{s}_{w1} = 0$. Selecting $\bar{s}_{w1} = (0, 1)^T$, we have

$$\bar{g}_w^T \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = 0 \Rightarrow \bar{g}_w^T \begin{pmatrix} 1 \\ 1 \end{pmatrix} = 0 \Rightarrow \bar{g}_w^T = (1, -1).$$

This means that this array should have a diagonal (skewed) memory layout. Note that, since the loop i is the innermost, this layout will maximize the locality for this array.

For the array U, we need to satisfy $\bar{p}^T \bar{s}_{u1} = 0$ and $\bar{g}_u^T \mathcal{L}_{u1} \bar{s}_{u1} = 0$. Selecting $\bar{s}_{u1} = (0, 1)^T$, we reach

$$\bar{g}_u^T \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = 0 \Rightarrow \bar{g}_u^T \begin{pmatrix} 1 \\ 0 \end{pmatrix} = 0 \Rightarrow \bar{g}_u^T = (0, 1),$$

meaning that the array U should be column-major.

The array V has both LHS and RHS references. We need to satisfy the following constraints:

$$\bar{p}^T \bar{s}_{v1} = 0, \quad \bar{g}_v^T \mathcal{L}_{v1} \bar{s}_{v1} = 0, \quad \text{and} \quad \bar{g}_v^T \mathcal{L}_{v2} \bar{s}_{v2} = 0.$$

Since $\bar{p}^T = (1, 0)$, the two options mentioned above are the same. Consequently, we select $\bar{s}_{v1} = \bar{s}_{v2} = (0, 1)^T$ and obtain the following two equations:

$$\bar{g}_v^T \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = 0 \quad \text{and} \quad \bar{g}_v^T \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = 0,$$

respectively, for the two references to array V. Therefore, from

$$\bar{g}_v^T \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = 0 \Rightarrow \bar{g}_v^T \begin{pmatrix} 0 \\ 1 \end{pmatrix} = 0 \Rightarrow \bar{g}_v^T = (1, 0),$$

we find that the layout of V should be row-major. To sum up, the layouts of the arrays U, V, and W should be column-major, row-major, and diagonal, respectively. These layouts help us to exploit locality in the innermost loop while reducing the amount of potential false sharing.

We next move to the second nest. Now, suppose that, due to possible dependences arising from references to arrays other than X, only the innermost loop k can be parallel, meaning that $\bar{p}^T = (0, 0, 1)$. Let us concentrate on array X. The two access matrices are

$$\mathcal{L}_{x1} = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix} \text{ and } \mathcal{L}_{x2} = \begin{pmatrix} 1 & 1 & 0 \\ 0 & -1 & 1 \end{pmatrix}.$$

We have three constraints to be satisfied:

7. To be specific, the global strategy just omits the arrays whose layouts have already been determined and focuses on the remaining arrays.

TABLE 1
Programs in Our Experimental Suite and Their Important Characteristics

program no.	code	source	size	iter	no. of arrays	max dim	type	over LOL	over LOU
1	hydro2d/T1	spec92	4,000	10	4	2	benchmark	16.4	4.7
2	hydro2d/fct	spec92	4,000	2	20	2	benchmark	17.0	6.9
3	vpenta	spec92	920	10	9	3	benchmark	24.4	7.0
4	emit	spec92	4,000	5	16	2	benchmark	19.1	7.0
5	btrix	spec92	820	5	29	4	benchmark	18.2	7.1
6	mxm	spec92	2,048	10	3	2	benchmark	20.8	8.8
7	cholsky	spec92	1,800	5	3	3	benchmark	42.0	2.0
8	gmtry	spec92	2,250	5	6	2	benchmark	34.7	3.5
9	adi	livermore	3,000	1	6	3	benchmark	15.6	2.2
10	bakvec	eispack	4,000	20	3	2	library	25.1	8.1
11	htribk	eispack	2,048	2	5	2	library	23.1	7.4
12	qzhes	eispack	4,000	5	3	2	library	28.6	7.5
13	fnorm	odepack	4,000	5	3	2	library	14.6	6.0
14	gfunp	hompack	4,000	12	6	2	library	14.9	6.4
15	r1mpyq	minpack	4,000	12	1	2	library	19.3	8.0
16	transpose	nwchem	4,096	10	2	2	application	19.9	9.0
17	hnd_nw_hnd	nwchem	4,096	10	8	3	application	28.5	7.9
18	hnd_nwhnd_tran	nwchem	4,096	5	10	2	application	40.6	8.0
19	hnd_int_1e_studd	nwchem	4,096	5	4	4	application	25.2	8.2
20	hnd_whermt	nwchem	4,096	2	6	3	application	13.0	4.3

$$\bar{p}^T\bar{s}_{x1}=0,\quad \bar{g}_x^T\mathcal{L}_{x1}\bar{s}_{x1}=0,\quad \text{and}\quad \bar{g}_x^T\mathcal{L}_{x2}\bar{s}_{x2}=0.$$

Considering the first option, we can set $\bar{s}_{x1}=(0,1,0)^T$ and $\bar{s}_{x2}=(0,0,1)^T$. Then,

$$\bar{g}_x^T\begin{pmatrix}1&0&1\\0&1&1\end{pmatrix}\begin{pmatrix}0\\1\\0\end{pmatrix}=0\Rightarrow\bar{g}_x^T\begin{pmatrix}0\\1\end{pmatrix}=0\Rightarrow\bar{g}_x^T=(1,0),$$

which implies that the layout of X should be row-major. Similarly, from

$$\bar{g}_x^T\begin{pmatrix}1&1&0\\0&-1&1\end{pmatrix}\begin{pmatrix}0\\0\\1\end{pmatrix}=0\Rightarrow\bar{g}_x^T\begin{pmatrix}0\\1\end{pmatrix}=0\Rightarrow\bar{g}_x^T=(1,0),$$

we also find that the preferred layout is row-major. Therefore, in this option, we are able to satisfy both the constraints by selecting a row-major memory layout for the array X.

We now focus on the second option. We try to satisfy the two constraints by setting $\bar{s}_{x1}=\bar{s}_{x2}=(0,0,1)^T$. Therefore,

$$\bar{g}_x^T\begin{pmatrix}1&0&1\\0&1&1\end{pmatrix}\begin{pmatrix}0\\0\\1\end{pmatrix}=0\Rightarrow\bar{g}_x^T\begin{pmatrix}1\\1\end{pmatrix}=0\Rightarrow\bar{g}_x^T=(1,-1),$$

which implies that the layout of X should be diagonal. On the other hand, from

$$\bar{g}_x^T\begin{pmatrix}1&1&0\\0&-1&1\end{pmatrix}\begin{pmatrix}0\\0\\1\end{pmatrix}=0\Rightarrow\bar{g}_x^T\begin{pmatrix}0\\1\end{pmatrix}=0\Rightarrow\bar{g}_x^T=(1,0),$$

we find that the second constraint prefers a row-major layout. Assuming a profile information prefers one reference (constraint) over the other, using this option, we can satisfy only a single constraint.

Comparing these two options, it is clear that the first option is preferable, therefore, we set the layout of the array X to row-major.

5 Experimental Results

We now present preliminary experimental results obtained on an eight-processor SGI/Cray Origin 2,000 distributed shared memory multiprocessor [37] at Northwestern University. The SGI Origin uses R10K processors, each of which is a 4-way superscalar microprocessor operating at a clock frequency of 195 MHz. Each processor has a 32 KB on-chip instruction cache and can issue instructions to its four functional units out-of-order. It also has a 32 KB two-way set-associative on-chip data cache and a 4 MB unified external cache, which are called the primary and the secondary cache, respectively. The latency ratio between the first-level (on-chip) data cache, second-level (off-chip) cache, and main memory is approximately $1:5:60$. The cache line size is 128 bytes and the page size is 16 KB. The local memory of each node (which consists of two processors) is made up of 128 MB SDRAM.

We conducted extensive experiments to measure the impact of our approach on locality and false sharing using 20 programs from different domains (benchmarks, library codes, and real application codes). Table 1 gives the important features of these codes. The size column gives (in terms of double precision elements) the maximum size of a dimension of any array used in the program and the iter column shows how many times the outermost timing loop has been executed for each code. The max-dim column on the other hand, shows the maximum dimensionality of any array used in the respective code. The last two columns will be explained later.

When necessary (to eliminate a constraint during optimization), we used profile data to select the constraint to be dropped from consideration. Since our constraints (false sharing or locality) are obtained from array references, we can easily associate a constraint with one or more array references. Consequently, eliminating a constraint can be reexpressed as eliminating an array reference(s). To measure how important each array reference is, we instrumented each code to keep track of how many times each array reference is touched during execution. When this

TABLE 2
Different Versions Used in Our Experiments

version	brief description	references
ORI	original (unoptimized) code with column-major memory layouts for all arrays	
LOL	locality optimized version using loop (iteration space) transforms only	native compiler, [39], [29]
LOD	locality optimized version using data (memory layout) transforms only	[38], [42], [26]
LOU	locality optimized version using both loop and data transforms	[28], [27], [12]
FOL	false sharing optimized version using loop (iteration space) transforms only	authors, [18], [6]
FOD	false sharing optimized version using data (memory layout) transforms only	authors, [46], [22]
FOU	false sharing optimized version using both loop and data transforms	authors, [13], [18]
BAL	version obtained using the approach discussed in this paper	authors
HND	hand optimized version using both linear and non-linear transforms	authors

instrumented version is executed, we can order the references in the code according to their importance. Using this profiling strategy, the references in inner loop levels are considered more important than the references in outer loop levels. Similarly, the references in loops with large trip counts (number of iterations) are considered more important than the references in loops with small trip counts. We also performed experiments with different input data (keeping the data size fixed). However, since in array-intensive codes the runtime execution path is largely determined by loop bounds and array sizes (rather than specific values of input arrays), we did not observe any significant dependence on input. Even with smaller input sizes, the relative importance of different array references with respect to each other did not change too much.

Table 2 shows the *versions* used in our experiments. The optimized versions can be divided into several groups. In the first group consisting of LOL, LOD, and LOU, the optimizations used are aimed only at enhancing spatial locality. Among these versions, LOU is the most powerful as it employs both loop and data transformations to improve locality. In the second group (consisting of FOL, FOD, and FOU), the optimizations attempt to reduce false sharing rather than optimizing spatial locality exclusively, therefore, they are most useful on multiprocessors. In this group, the most powerful technique is FOU, which uses both loop and data transformations for minimizing false sharing. For each version, we tried to use as many published techniques as possible (listed under the referencescolumn in Table 2) and selected the one that performs best.[8] In all these six versions, only *linear* loop and data transformations were used. For example, tiling or loop unrolling [51] were not used. The BAL version refers to the approach discussed in this paper, and HND is the hand-optimized version using *both* linear and nonlinear (e.g., tiling) loop and data transformations. Note that, with the hand-optimized version (HND), we did not pay great attention to choosing tile sizes; the use of tile size selection heuristics [14], [34] may further improve the performance of the HND version.

8. Our own approaches [26], [28] as well as the heuristic presented in this paper are implemented using Parafrase-2 [43]. For the approaches of other researchers, we hand coded the optimizations relying on the explanations in the respective papers; also, in those cases, we used only the techniques that are applicable to array-based codes. For example, from [18] we used only loop and array alignment, from [46] we used only array alignment, and from [13] we used software caching where applicable. For each version, we selected the result of the best performing version. As an example, the LOL version used the best of the output codes of native compiler [29] and [39]. The entry authors denotes our own techniques that have not been published yet. Also, note that all array dimensions were padded [45], [2] by a small amount to eliminate power-of-two sizes.

For all the versions, before transforming the code for locality and/or false sharing, we detected the largest granularity parallelism using the native Fortran compiler (MIPSpro version 7.20) with locality optimizations turned off.[9] Note that FOL, FOD, FOU, BAL, and HND take parallelism decisions explicitly into account, whereas LOL, LOD, and LOU can reduce false sharing only as a side effect of improving locality. After the different versions were obtained, we again used the native compiler (with the -O2 option and all the scalar optimizations turned on) to generate the executables.

The results for single processor case are presented in Table 3, and the results for the eight processor case are shown in Table 4. In these tables, the second column (ORI) gives the total execution times in *seconds*. Columns three through 10 show the *percentage improvement* obtained by using the respective versions *over* ORI. The improvement here means *reduction* in the overall execution time and a negative entry indicates an increase in the execution time with respect to ORI.

From Table 3, we can conclude the following: The pure locality oriented techniques are able to improve the overall performance significantly. In 14 of our 20 programs, the LOD version had the opportunity for applying some kind of data transformation. Similarly, in 12 of our codes, the LOL version chose to apply some kind of loop transformation. We observe that in nine programs the LOU and FOU version generated the same results. We also observe that LOU brings on average 23.44 percent improvement over the original codes. However, three is a large variance between applications. For example, btrix and vpenta show very large improvements (mostly due to layout transformations). The btrix benchmark, for instance, accesses four-dimensional arrays and the original access pattern is very poor. Loop transformations eliminate some of the problems, but in particular, two loop nests in this code cannot be fully optimized due to data dependences. A framework that uses both loop and data transformations in concert on the other hand, achieve large performance benefits. In contrast, in hydro2d/fct, mxm, cholsky, qzhes, and hnd_int_1e_studd, the default memory layout is appropriate for the default access pattern. For example, in mxm, the original code is both unrolled and loop transformed for locality. Consequently, neither loop nor data transformations are effective. In addition, we also observe that in some cases data transformations actually degrade performance. This is because in some cases the compiler can decide to apply a data transformation,

9. The locality optimizations are turned on for generating the LOL version.

TABLE 3
Results On a Single Processor

program no.	`ORI` (sec)	`LOL` (%)	`LOD` (%)	`LOU` (%)	`FOL` (%)	`FOD` (%)	`FOU` (%)	`BAL` (%)	`HND` (%)	*imprv1* (%)	*imprv2* (%)
1	128.13	0.00	0.00	2.51	0.00	0.00	0.00	7.86	8.65	5.35	0.79
2	53.80	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
3	30.50	0.00	80.32	84.30	0.00	0.00	0.00	90.04	90.04	5.74	0.00
4	10.13	5.40	5.00	8.16	−2.25	−4.00	−2.25	22.66	22.66	14.50	0.00
5	56.54	12.35	86.41	90.85	8.30	−3.38	10.67	90.85	90.85	0.00	0.00
6	119.86	4.18	−11.40	4.18	4.18	4.18	4.18	4.18	34.13	0.00	29.95
7	71.18	−4.55	0.00	0.00	−6.00	0.00	0.00	0.00	9.90	0.00	9.90
8	58.40	0.00	0.00	0.00	0.00	0.00	0.00	0.00	2.65	0.00	2.65
9	34.24	56.24	33.59	56.24	12.00	0.00	12.30	66.04	67.11	9.80	1.07
ave. (B)	62.53	8.18	21.55	27.36	1.80	−0.36	2.77	31.29	36.22	3.93	4.93
10	43.13	4.05	4.28	21.00	−10.36	−7.40	−7.40	43.36	43.36	22.36	0.00
11	24.05	−38.00	45.00	48.95	−38.00	−36.60	−36.60	48.95	50.16	0.00	1.21
12	21.00	0.00	−1.00	0.00	0.00	−2.00	0.00	0.00	20.02	0.00	20.02
13	10.37	68.54	61.10	68.54	68.54	14.82	68.54	68.54	68.54	0.00	0.00
14	88.15	0.00	0.00	14.54	0.00	0.00	0.00	16.02	22.84	1.48	6.82
15	12.50	−2.51	0.00	5.80	0.00	0.00	0.00	9.65	9.65	3.85	0.00
ave. (L)	33.20	5.35	18.23	26.47	3.63	−5.20	4.09	31.09	35.76	4.62	4.67
16	32.14	0.00	35.18	35.18	0.00	35.18	35.18	35.18	44.11	0.00	8.93
17	47.66	2.36	3.40	3.40	2.36	3.40	3.40	3.40	6.00	0.00	2.60
18	24.10	4.40	5.21	5.21	−10.05	5.21	5.21	5.21	5.21	0.00	0.00
19	24.23	0.00	−4.86	0.00	−4.41	−4.41	−4.41	24.33	30.44	24.33	6.11
20	16.55	18.96	20.03	20.03	18.96	18.96	18.96	20.03	41.00	0.00	20.97
ave. (A)	28.94	5.14	11.79	12.76	1.37	11.67	11.67	17.63	25.35	4.87	7.72
ave.	45.33	5.57	18.11	23.44	2.16	1.20	5.39	27.82	33.37	4.37	5.55

ave. (B), ave. (L), and ave. (A) denote the averages for the benchmark codes, library codes, and application codes, respectively.

however, since a data transformation affects the locality behavior of all references in the code, it can create a negative overall impact. Similarly, as in `cholsky`, `qzhes`, and `r1mpyq`, in some cases, loop transformations make the subscript expressions very complex and back-end optimizations may not be able to eliminate the performance overhead due to these complex expressions. So, we observe some degradation in performance (as compared to the original code) when this occurs.

The false sharing oriented optimizations on the other hand, hardly achieve any improvement. In fact, the `FOD` version increases the execution time in benchmark and library codes. These poor results are due to the overhead of false sharing optimizations and the poor spatial locality they obtain in uniprocessor runs. In one benchmark (`transpose`), however, `FOD` improves performance significantly as the data transformations it applies are very suitable from the spatial locality perspective as well.

The `BAL` version performs very well, even in the uniprocessor case due to its ability to satisfy as many locality and false sharing constraints as possible. In particular, although not presented here in detail, 92 percent of the (dynamic) array references that are optimized by `LOU` are also optimized by `BAL`, indicating that taking false sharing constraints into account sacrifices little performance (as far as locality is concerned). The *imprv1* column gives the difference between `BAL` and the next best version from among the columns three through eight (usually `LOU`). This means that the `BAL` version outperforms a hypothetical approach that applies all these techniques and selects the best one. Notice that, as compared to the classical loop transformation techniques (`LOL`), `BAL` obtains more than 22 percent further reduction.

It should be noted that, even in single processor case, the `BAL` version performs better than the `LOU` version. While this is counter-intuitive (as it is not clear why a strategy that takes false-sharing into account should outperform a pure locality-based approach on a single processor), our further study shows that this is due to the global optimization strategy that our framework uses. What we have found is that a pure locality-oriented strategy determines the layouts earlier in the optimization process than a strategy that takes false-sharing into account. Consequently, with the former strategy, the layout constraints accumulate more quickly and the remaining nests have more restrictions in determining the most suitable layouts as far as their own (intranest) locality is concerned. The strategy that takes false-sharing into account, on the other hand, spreads the layout determination more evenly across different nests and achieves a better overall performance. In particular, with codes such as `bakvec` and `hnd_int_1e_studd`, if the array layouts are determined late in the optimization process, much better results are achieved. It should be emphasized, however, that this observation may be specific to the codes in our experimental suite and, for some other codes, we can expect the `LOU` version to outperform the `BAL` version as far as the uniprocessor performance is concerned.

The *imprv2* column shows the difference between `HND` and `BAL`. Except for three programs, the additional improvement obtained by hand optimization over `BAL` is always below 10 percent. This additional improvement comes almost exclusively from loop tiling. For example, in `mxm`, tiling the innermost two loops makes a significant difference in performance. Similarly, in `gfunp`, tiling the most time consuming nest increases the performance by

TABLE 4
Results on Eight Processors

program no.	ORI (sec.)	LOL (%)	LOD (%)	LOU (%)	FOL (%)	FOD (%)	FOU (%)	BAL (%)	HND (%)	*imprv1* (%)	*imprv2* (%)
1	21.72	0.00	0.00	2.30	0.00	0.00	0.00	9.11	9.86	6.81	0.75
2	9.61	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
3	10.89	0.00	63.21	64.40	0.00	0.00	0.00	87.17	87.17	22.77	0.00
4	3.27	2.90	5.05	7.05	4.90	6.33	6.90	30.01	30.01	22.96	0.00
5	20.19	13.07	55.91	70.04	19.16	−1.15	24.35	70.04	70.04	0.00	0.00
6	21.03	3.90	−16.51	4.40	8.21	8.21	8.21	9.87	24.55	1.66	14.68
7	16.95	−8.21	0.00	0.00	−9.33	0.00	0.00	0.00	10.86	0.00	10.86
8	13.90	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.08	0.00	1.08
9	5.90	40.00	24.61	40.00	7.75	0.00	7.75	65.16	71.04	25.16	5.88
ave. (B)	13.72	5.74	14.70	20.91	3.41	1.49	5.25	30.15	33.85	9.24	3.70
10	10.27	2.05	4.20	15.65	4.49	6.91	6.91	40.86	40.86	25.21	0.00
11	9.25	−27.02	42.24	44.00	−11.00	−7.14	−7.14	44.00	51.07	0.00	7.07
12	7.50	0.00	−1.00	0.00	0.00	8.95	0.00	8.95	14.69	0.00	5.74
13	2.21	68.90	62.33	68.90	68.90	16.80	68.90	68.90	68.90	0.00	0.00
14	17.28	0.00	0.00	10.11	0.00	0.00	0.00	20.04	22.67	9.93	2.63
15	2.55	−7.66	0.00	5.13	0.00	0.00	0.00	8.00	8.00	2.87	0.00
ave. (L)	8.18	6.05	17.96	23.97	10.40	4.25	11.45	31.79	34.37	7.82	2.58
16	5.27	0.00	20.28	20.28	0.00	20.28	20.28	20.28	34.62	0.00	14.34
17	9.17	2.30	3.15	3.15	2.30	3.15	3.15	3.15	5.55	0.00	1.40
18	3.39	4.05	5.50	5.50	4.46	19.81	19.81	34.63	47.89	14.82	13.26
19	6.21	0.00	−9.50	0.00	13.21	13.21	13.21	29.85	49.55	16.69	19.70
20	3.38	15.21	19.85	19.85	15.21	15.21	15.21	19.85	40.86	0.00	21.01
ave. (A)	5.48	4.31	7.86	9.76	7.04	14.33	14.33	21.55	35.69	7.22	14.14
ave.	9.99	5.47	13.97	19.04	6.41	5.53	9.38	28.49	34.46	9.01	5.97

ave. (B), ave. (L), and ave. (A) denote the averages for the benchmark codes, library codes, and application codes, respectively.

nearly 30 percent. These results motivate us for further research on combining iteration space tiling with our optimization framework.

In the case of eight processors (Table 4), as expected, the false sharing oriented techniques perform better than they do for the uniprocessor case. In particular, FOU is now able to reduce the original execution times by nearly 9 percent. However, this is still below the performance of LOU (which is around 19 percent). This is consistent with the conclusion of Torrellas et al. [46], that reducing false sharing at all costs is not a good idea. We also note that the impact of pure locality-based approaches is decreased when we move from one processor to eight processors, mostly due to a reduction in the working set sizes per processor and due to not considering parallelism decisions in determining the optimal memory layouts. For example, in transpose, not taking parallelism (false sharing) constraints into account results in selecting different memory layouts for the two most active arrays (as compared to the case when these constraints are accounted for). A similar scenario occurs in htribk as well. The BAL version, on the other hand, does take the parallelism decisions into account and achieves a 28.49 percent improvement on the average; it reduces memory system delays, decreases the working set size per processor, and minimizes the coherence overhead in multiprocessor runs. By including hand optimizations, we can get an additional 6 percent benefit, most of which can be obtained by attributed to a judicious application of tiling (i.e., tiling only the loops that carry some reuse [49]) after using BAL. In particular, the 14.14 percent performance gap between BAL and HND in application codes encourages us to use control-centric tiling [21] and data-centric tiling [32] once our approach has been applied. An in-depth understanding of the interaction between our approach and tiling, however, merits further study and is outside the scope of this paper.

We also need to say that using BAL increased the total compilation time on the average by 23 percent over LOL, and increased it by 6.5 percent over LOU (The last two columns of Table 1 show the increases in compilation times (in percent) over LOL and LOU, respectively, per program basis). Finally, on the average, there was only a 7 percent increase on the total memory requirements of the codes after data transformations (most of which in the application codes). Given the benefits of our technique, we believe that the reasonable increases in compilation time and memory requirements are bearable.

In general, we can expect that our unified strategy would still be effective if we had only a single-level cache hierarchy (instead of our two-level cache hierarchy). The array (page) distribution strategy can make some difference in the results (absolute execution times) we obtain. In all results presented in this section, we used the round-robin page allocation strategy. This is reasonable as this strategy tries to distribute the data pages evenly across processors and balances the workload. We activated this page allocation strategy by inserting a compiler directive (provided by SGI) at the beginning of each code. An alternative strategy would be the first-touch page allocation strategy, where a page is stored in memory of the processor that touches it first. However, in codes in our experimental suite, the array initialization loops are not always parallelized. Consequently, the first-touch page allocation strategy would generate much worse execution times by overloading the memory of a single processor. However, we expect this degradation to be valid for all different versions used in our experiments. In particular, we expect that the BAL version would still generate the best results. Studying the interaction between our optimizations and page allocation strategy is a topic that we will revisit in future.

6 Related Work

Compiler researchers have attacked the locality optimization problem from several points of view. Most of the research has focused on enhancing the cache locality of scientific computations using loop transformations. Wolf and Lam [49] presented formal definitions of several types of reuse and offered a framework that uses unimodular loop transformations as well as tiling. Li [39] also focused on cache locality, but considered general nonsingular loop transformation matrices. McKinley et al. [41] presented a simple algorithm that unifies loop permutation, fusion, and distribution. Other researchers have also considered tiling [14], [21], [32], [49], [9]. All of these approaches use only iteration space transformations and, consequently, they are constrained by intrinsic data dependences in the program. Since it might be difficult to find a loop transformation that satisfies all the references in a loop nest, these approaches are limited in their ability to improve locality for all the arrays referenced in a nest. Moreover, since most of these techniques are specifically for optimizing the performance of uniprocessor caches, they do not take false sharing into account.

Recently, techniques based on memory layout transformations for improving locality have been proposed. Leung and Zahorjan [38], O'Boyle and Knijnenburg [42], and Kandemir et al. [26], [25] proposed techniques that change memory layouts. Although such techniques can improve the spatial locality characteristics of the programs significantly, they may not be as effective on multiprocessors due to false sharing as they do not take parallelism information into account. In contrast, Cierniak and Li [12] and Kandemir et al. [27] offered techniques that employ both loop and data transformations to improve locality. Besides suffering from disadvantages of loop transformations, these techniques also suffer from the effects of false sharing in those cases where outermost loop parallelism is not available. Anderson et al. [1] also propose data transformations to improve locality and eliminate false sharing; they use permutations and strip-mining for possible data transformations. Our work is more general as we consider a larger search space for possible layout transformations.

Kennedy and McKinley [31] explore the tradeoffs between effectively utilizing parallelism and memory hierarchy on shared memory parallel machines. They use strip-mining and loop permutation in order to exploit both parallelism and data locality. There is also considerable work on reducing false sharing in shared-memory parallel machines. Torrellas et al. [46] applied a number of data transformations such as array padding and block alignment to eliminate false sharing. They hypothesize that false sharing is not the major source of cache misses on shared-memory machines; instead, most of the misses are due to poor spatial locality. However, they offer no systematic approach that can be automated for balancing spatial locality and false sharing for array-based codes. Jeremiassen and Eggers [22], [15] also proposed data transformations to reduce false sharing. Their optimizations either group data that is accessed by the same processor or separate individual data items that are shared. Although some of their transformations help improve spatial locality, others may adversely affect locality. In comparison, we focus more on structured codes and demonstrate how spatial locality and false sharing can be treated in an optimizing compiler framework.

Bianchini and LeBlanc [5], Eggers and Katz [17], and Ju and Dietz [23] also observe the impact of false sharing on parallel programs and propose several techniques to manage it. None of these works explicitly studied the interaction between optimizing locality and reducing false sharing. In contrast, Bolosky et al. [8] proposed coalescing different data into a larger data set and padding data to page boundaries to eliminate false sharing at the page level. Since padding to page boundaries can be very expensive and distorts spatial locality, it is not clear to us how successful this method will be on modern cache-coherent architectures. Granston and Wijshoff [18] discussed loop and data transforms for eliminating false sharing in shared virtual memory systems, however, they do not propose a complete methodology and no experimental results are presented. Bodin et al. [6], [7] also proposed loop transformation techniques for reducing page-level multiple-writer false sharing. However, they do not investigate the interaction between parallelism decisions, spatial locality, and false sharing. Another drawback of the previous work on false sharing specific optimizations is that, in reality, the number of processors available for a run may not be known at compile time. If it happens to be just one processor, then all these false sharing-oriented page-level transformations may introduce significant overhead at runtime. In comparison, we take a more balanced view of the problem and do not eliminate false sharing at all costs. Cierniak and Li [13] proposed software caching and dynamic layout modifications to reduce false sharing. They found that false sharing optimizations improve spatial locality as well. Their results can be attributed to the available outermost loop parallelism in the small kernels that they used. Finally, Chow and Sarkar [11] proposed the modification of runtime scheduling parameters for eliminating multiple-writer false sharing. We believe that their solution is complementary to our approach.

Finally, there is some work on developing cache-conscious applications and scientific libraries. Frens and Wise [16] give a recursive matrix-multiply algorithm that uses nonlinear array layouts. LaMarca and Ladner investigate the influence of caches on the performance of heaps [35] and present a cache performance analysis of well-known algorithms [33]. Bilmes et al. present a locality-optimized matrix-multiply routine [4]. Chatterjee et al. [10] consider nonlinear memory layouts for optimizing cache locality in some matrix computations.

7 Summary and Future Work

The performance of programs on current shared-memory multiprocessors with coherent caches depends on several factors such as the interaction between the granularity of data sharing, the size of the coherence unit, and the spatial locality exhibited by the applications, in addition to the amount of parallelism in the applications. In this paper, we have presented a mathematical framework for studying the interaction between false sharing and locality for programs on shared-memory multiprocessors. We have found that, in those cases where the compiler can obtain outermost loop parallelism, it is possible to simultaneously enhance spatial locality and reduce false sharing using memory layout transformations, while honoring the parallelism decisions already made by the compiler. On a collection of 20 programs drawn from various sources, the balanced approach presented in this paper, brings about an additional 9 percent improvement over powerful loop and data transformations aimed specifically at locality and shows a 19 percent improvement over techniques aimed only at reducing false sharing. This clearly demonstrates the benefits of balancing locality and false sharing. In those cases where outermost loop parallelism is not possible, we need to favor eliminating false sharing over optimizing locality or vice-versa; detailed profile informations might be useful in making this decision.

In the future, we plan to embed loop transformations (other than those aimed only at deriving parallel loops) into our framework. Although, in principle, such a framework should be more powerful than the one discussed in this paper; including loop transformations brings a number of other issues into the picture. In addition, we plan to enhance our data transformation framework so that it can work in an interprocedural setting where the arrays may be reshaped across procedure boundaries.

Acknowledgments

The authors would like to thank the anonymous referees for providing helpful comments. The material presented in this paper is based on research supported in part by the US National Science Foundation grants CCR-9357840 and CCR-9509143, and the Air Force Materials Command under contract F30602-97-C-0026. P. Banerjee is supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract F30602-98-2-0144. J. Ramanujam is supported in part by a US National Science Foundation Young Investigator Award CCR-9457768, a US National Science Foundation grant CCR-0073800, and a US National Science Foundation Information Technology Research grant CHE-0121706.

References

[1] J. Anderson, S. Amarasinghe, and M. Lam, "Data and Computation Transformations for Multiprocessors," *Proc. Fifth ACM SIGPLAN Symp. Principles and Practice of Parallel Programming,* pp. 166-178, July 1995.

[2] D.F. Bacon, J. Chow, D.R. Ju, K. Muthukumar, and V. Sarkar, *Proc. CASCON '94 Conf.,* Nov. 1994.

[3] U. Banerjee, "Unimodular Transformations of Double Loops," *Advances in Languages and Compilers for Parallel Processing,* A. Nicolau et al., eds., MIT Press, 1991.

[4] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel, "Optimizing Matrix-Multiply Using PHiPAC: A Portable, High-Performance ANSI C Coding Methodology," *Proc. Int'l Conf. Supercomputing,* pp. 340-347, July 1997.

[5] R. Bianchini and T. LeBlanc, "Software Caching on Cache-Coherent Multiprocessors," *Proc. Fourth IEEE Symp. Parallel and Distributed Processing,* Dec. 1992.

[6] F. Bodin, E. Granston, and T. Montaut, "Evaluating Two Loop Transformations for Reducing Multiple-Writer False Sharing," *Proc. Seventh Ann. Workshop Languages and Compilers for Parallel Computing,* Aug. 1994.

[7] F. Bodin, E. Granston, and T. Montaut, "Page-Level Affinity Scheduling for Eliminating False Sharing," *Proc. Fifth Workshop Compilers for Parallel Computing,* June 1995.

[8] W. Bolosky, R. Fitzgerald, and M. Scott, "Simple But Effective Techniques for NUMA Memory Management," *Proc. 12th ACM Symp. Operating Systems Principles,* Dec. 1989.

[9] L. Carter, J. Ferrante, S. Flynn Hummel, B. Alpern, and K. Gatlin, "Hierarchical Tiling: A Methodology for High Performance," *UCSD Technical Report CS96-508,* Nov. 1996.

[10] S. Chatterjee, V.V. Jain, A.R. Lebeck, S. Mundhra, and M. Thottethodi, "Nonlinear Array Layouts for Hierarchical Memory Systems," *Proc. ACM Int'l Conf. Supercomputing (ICS '99),* June 1999.

[11] J. Chow and V. Sarkar, "False Sharing Elimination by Selection of Runtime Scheduling Parameters," *Proc. 26th Int'l Conf. Parallel Processing,* Aug. 1997.

[12] M. Cierniak and W. Li, "Unifying Data and Control Transformations for Distributed Shared Memory Machines," *Proc. SIGPLAN '95 Conf. Programming Language Design and Implementation,* pp. 205-217, June 1995.

[13] M. Cierniak and W. Li, "A Practical Approach to the Compile-Time Elimination of False Sharing for Explicitly Parallel Programs," *Proc. 10th Ann. Int'l Conf. High Performance Computers,* June 1996.

[14] S. Coleman and K. McKinley, "Tile Size Selection Using Cache Organization and Data Layout," *Proc. SIGPLAN '95 Conf. Programming Language Design and Implementation,* June 1995.

[15] S. Eggers and T. Jeremiassen, "Eliminating False Sharing," *Proc. Int'l Conf. Parallel Processing,* vol. I, pp. 377-381, Aug. 1991.

[16] J.D. Frens and D.S. Wise, "Auto-Blocking Matrix Multiplication or Tracking BLAS3 Performance with Source Code," *Proc. Sixth ACM SIGPLAN Symp. Principles and Practice of Parallel Programming,* pp. 206-216, June 1997.

[17] S. Eggers and R. Katz, "The Effect of Sharing on the Cache and Bus Performance of Parallel Programs," *Proc. Third Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* pp. 257-270, Apr. 1989.

[18] E. Granston and H. Wijshoff, "Managing Pages in Shared Virtual Memory Systems: Getting the Compiler Into the Game," *Proc. Int'l Conf. Supercomputing,* pp. 11-20, 1993.

[19] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 1995.

[20] C.-H. Huang and P. Sadayappan, "Communication-Free Partitioning of Nested Loops," *J. Parallel and Distributed Computing,* vol. 19, pp. 90-102, 1993.

[21] F. Irigoin and R. Triolet, "Supernode Partitioning," *Proc. 15th Ann. ACM Symp. Principles of Programming Languages,* pp. 319-329, Jan. 1988.

[22] T. Jeremiassen and S. Eggers, "Reducing False Sharing on Shared Memory Multiprocessors Through Compile Time Data Transformations," *Proc. Fifth ACM SIGPLAN Symp. Principles and Practice of Parallel Programming,* July 1995.

[23] Y. Ju and H. Dietz, "Reduction of Cache Coherence Overhead by Compiler Data Layout and Loop Transformation," *Proc. Workshop Languages and Compilers for Parallel Computing,* U. Banerjee et al., eds., pp. 344-358, 1992.

[24] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee, "A Graph-Based Framework to Detect Optimal Memory Layouts for Improving Data Locality," *Proc. 1999 Int'l Parallel Processing Symp.,* Apr. 1999.

[25] M. Kandemir, A. Choudhary, N. Shenoy, P. Banerjee, and J. Ramanujam, "A Linear Algebra Framework for Automatic Determination of Optimal Data Layouts," *IEEE Trans. Parallel and Distributed Systems,* vol. 10, no. 2, Feb. 1999.

[26] M. Kandemir, A. Choudhary, N. Shenoy, P. Banerjee, and J. Ramanujam, "A Hyperplane Based Approach for Optimizing Spatial Locality in Loop Nests," *Proc. 1998 ACM Int'l Conf. Supercomputing,* pp. 69-76, July 1998.

[27] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee, "Improving Locality Using Loop and Data Transformations in an Integrated Framework," *Proc. 31st Ann. ACM/IEEE Int'l Symp. Microarchitecture,* vol. 31, pp. 285-296, Dec. 1998.

[28] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee, "A Matrix-Based Approach to the Global Locality Optimization Problem," *Proc. 1998 Int'l Conf. Parallel Architectures and Compilation Techniques,* Oct. 1998.

[29] M. Kandemir, J. Ramanujam, A. Choudhary, and P. Banerjee, "An Iteration Space Transformation Algorithm Based on Explicit Data Layout Representation for Optimizing Locality," *Proc. Workshop Languages and Compilers for Parallel Computing,* Aug. 1998.

[30] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott, "The Omega Library Interface Guide," Technical Report CS-TR-3445, CS Dept., Univ. of Maryland, College Park, Mar. 1995.

[31] K. Kennedy and K.S. McKinley, "Optimizing for Parallelism and Data Locality," *Proc. 1992 ACM Int'l Conf. Supercomputing (ICS '92),* July 1992.

[32] I. Kodukula, N. Ahmed, and K. Pingali, "Data-Centric Multi-Level Blocking," *Proc. SIGPLAN '97 Conf. Programming Language Design and Implementation,* June 1997.

[33] R. Ladner, J. Fix, and A. LaMarca, "Cache Performance Analysis of Algorithms," *Proc. 10th Ann. ACM-SIAM Symp. Discrete Algorithms,* Jan. 1999.

[34] M. Lam, E. Rothberg, and M. Wolf, "The Cache Performance of Blocked Algorithms," *Proc. Fourth Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS '91),* Apr. 1991.

[35] A. LaMarca and R. Ladner, "The Influence of Caches on the Performance of Heaps," *The ACM J. Experimental Algorithms,* vol. 1, 1996.

[36] L. Lamport, "The Parallel Execution of DO Loops," *Comm. ACM,* vol. 17, no. 2, pp. 83-93, Feb. 1974.

[37] J. Laudon and D. Lenoski, "The SGI Origin: A CC-NUMA Highly Scalable Server," *Proc. 24th Ann. Int'l Symp. Computer Architecture (ISCA '97),* May 1997.

[38] S.-T. Leung and J. Zahorjan, "Optimizing Data Locality by Array Restructuring," Technical Report TR 95-09-01, Dept. of Computer Science and Eng., Univ. of Washington, Sept. 1995.

[39] W. Li, "Compiling for NUMA Parallel Machines," PhD thesis, Cornell Univ., Ithaca, New York, 1993.

[40] W. Li and K. Pingali, "Access Normalization: Loop Restructuring for NUMA Compilers," *ACM Trans. Computer Systems,* vol. 11, no. 4, pp. 353-375, 1993.

[41] K. McKinley, S. Carr, and C.W. Tseng, "Improving Data Locality with Loop Transformations," *ACM Trans. Programming Languages and Systems,* vol. 18, no. 4, pp. 424-453, July 1996.

[42] M. O'Boyle and P. Knijnenburg, "Non-Singular Data Transformations: Definition, Validity, Applications," *Proc. Sixth Workshop Compilers for Parallel Computers,* pp. 287-297, 1996.

[43] C. Polychronopoulos, M. Girkar, M. Haghighat, C. Lee, B. Leung, and D. Schouten, "Parafrase-2: An Environment for Parallelizing, Partitioning, Synchronizing, and Scheduling Programs on Multiprocessors," *Int'l J. High Speed Computing,* vol. 1, no. 1, 1989.

[44] J. Ramanujam and P. Sadayappan, "Compile-Time Techniques for Data Distribution in Distributed Memory Machines," *IEEE Trans. Parallel and Distributed Systems,* vol. 2, no. 4, pp. 472-482, Oct. 1991.

[45] G. Rivera and C.-W. Tseng, "Data Transformations for Eliminating Conflict Misses," *Proc. 1998 ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI '98),* June 1998.

[46] J. Torrellas, M. Lam, and J. Hennessey, "False Sharing and Spatial Locality in Multiprocessor Caches," *IEEE Trans. Computers,* vol. 43, no. 6, pp. 651-663, June 1994.

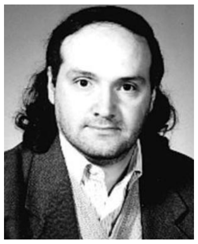
[47] E. Torrie, C-.W. Tseng, M. Martonosi, and M. Hall, "Evaluating the Impact of Advanced Memory Systems on Compiler-Parallelized Codes," *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques,* June 1995.

[48] C.-W. Tseng, J. Anderson, S. Amarasinghe, and M. Lam, "Unified Compilation Techniques for Shared and Distributed Address Space Machines," *Proc. 1995 Int'l Conf. Supercomputing,* July 1995.

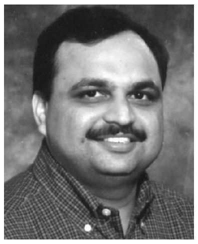
[49] M. Wolf and M. Lam, "A Data Locality Optimizing Algorithm," *Proc. ACM SIGPLAN '91 Conf. Programming Language Design and Implementation,* pp. 30-44, June 1991.

[50] M. Wolf and M. Lam, "A Loop Transformation Theory and an Algorithm to Maximize Parallelism," *IEEE Trans. Parallel and Distributed Systems,* vol. 2, no. 4, pp. 452-471, 1991.

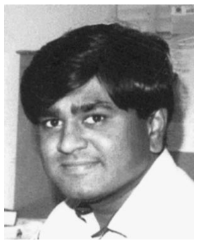
[51] M. Wolfe, *High Performance Compilers for Parallel Computing.* Addison Wesley, CA, 1996.



Mahmut Kandemir received the BSc and MSc degrees in control and computer engineering from Istanbul Technical University, Istanbul, Turkey, in 1988 and 1992, respectively. He received the PhD from Syracuse University, New York, in electrical engineering and computer science in 1999. He has been an assistant professor in the Computer Science and Engineering Department at the Pennsylvania State University since August 1999. His main research interests are optimizing compilers, I/O intensive applications, embedded systems, and power-aware computing. He is a member of the IEEE and the ACM.



Alok Choudhary received the PhD degree from the University of Illinois, Urbana-Champaign, in electrical and computer engineering in 1989, the MS degree from the University of Massachusetts, Amherst, in 1986, and the BE degree (Hons.) from the Birla Institute of Technology and Science, Pilani, India in 1982. He is a professor of electrical and computer engineering at Northwestern University. From 1993 to 1996 he was an associate professor in the Electrical and Computer Engineering Department at Syracuse University, and from 1989 to 1993 he was an assistant professor in the same department. He has worked in industry for computer consultants prior to 1984. Alok Choudhary received the US National Science Foundation's Young Investigator Award in 1993 (1993-1999). He has also received an IEEE Engineering Foundation award, an IBM Faculty Development award, and an Intel Research Council award. His main research interests are in high-performance computing and communication systems and their applications in many domains including multimedia systems, information processing, and scientific computing. In particular, his interests lie in the design and evaluation of architectures and software systems (from system software such as runtime systems, compilers, and programming languages to applications), high-performance servers, high-performance databases, and input-output. He has published more than 130 papers in various journals and conferences in the above areas. He has also written a book and several book chapters on the above topics. His research has been sponsored by (past and present) the Defense Advanced Research Projects Agency (DARPA), US National Science Foundation, NASA, Air Force Office of Scientific Research (AFOSR), Office of Naval Research (ONR), US Department of Energy (DOE), Intel, IBM, and TI. Alok Choudhary served as the conference cochair for the International Conference on Parallel Processing and as a program chair and general chair for the International Workshop on I/O Systems in Parallel and Distributed Systems. He also served a program vice-chair for HiPC '1999. He is an editor of the Journal of Parallel and Distributed Computing and an associate editor of IEEE Transactions on Parallel and Distributed Systems. He has also served as a guest editor for IEEE Computer and IEEE Parallel and Distributed Technology. He serves (or has served) on the program committee of many international conferences in architectures, parallel computing, multimedia systems, performance evaluation, distributed computing, etc. He is a member of the IEEE Computer Society and the ACM. He also serves in the High-Performance Fortran Forum, a forum of Academia, industry, and government labs working on standardizing programming languages for portable programming on parallel computers.



J. Ramanujam (Ram) received the BTech degree in electrical engineering from the Indian Institute of Technology, Madras, India, in 1983, and the MS and PhD degrees in computer science from The Ohio State University, Columbus, in 1987 and 1990, respectively. He is a professor of electrical and computer engineering at Louisiana State University, Baton Rouge. His research interests are in embedded systems, compilers for high-performance computer systems, software optimizations for low-power computing, high-level hardware synthesis, parallel architectures, and algorithms. He has published more than 90 papers in refereed journals and conferences in these areas in addition to several book chapters. Dr. Ramanujam received the US National Science Foundation's Young Investigator Award in 1994. He has served on the Program Committees of several conferences and workshops such as the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES '2001), the Workshop on Power Management for Real-Time and Embedded Systems, (IEEE Real-Time Applications Symposium, 2001), the International Conference on Parallel Architectures and Compilation Techniques (PACT 2000), the International Symposium on High Performance Computing (HiPC 99), and the 1997 International International Conference on Parallel Processing. He initiated and has coorganized the annual workshop on Compilers and Operating Systems for Low Power, which has been held in conjunction with PACT (Conference on Parallel Architectures and Compilation Techniques) since 2000. He has taught tutorials on compilers for high-performance computers at several conferences such as the International Conference on Parallel Processing (1998, 1996), Supercomputing 94, Scalable High-Performance Computing Conference (SHPCC 94), and the International Symposium on Computer Architecture (1993 and 1994). He has been a frequent reviewer for several journals and conferences. He is a member of the IEEE.

Prith Banerjee received the BTech degree in electronics and electrical engineering from the Indian Institute of Technology, Kharagpur, India, in August 1981, and the MS and PhD degrees in electrical engineering from the University of Illinois at Urbana-Champaign in December 1982 and December 1984, respectively. Dr. Banerjee is currently the Walter P. Murphy Professor and Chairman of the Department of Electrical and Computer Engineering, and Director of the Center for Parallel and Distributed Computing at Northwestern University in Evanston, Illinois. During 1985-1996, he was on the faculty of the University of Illinois. During that time, he was the Director of the Computational Science and Engineering program and Professor of Electrical and Computer Engineering and the Coordinated Science Laboratory at the University of Illinois at Urbana-Champaign. Prith Banerjee has also served as Founder, President, and CEO of a company called AccelChip during 2000-2002 while he was on leave from Northwestern University. This company was founded based on technology developed as part of a DARPA sponsored research on the MATCH compiler at Northwestern. Prith is continuing to serve as Chairman and Chief Technology Officer of AccelChip, Inc., in a part-time manner. His research interests are in parallel algorithms for VLSI design automation, distributed memory parallel compilers, and compilers for VLSI systems such as FPGAs and ASICs, and is the author of more than 300 papers in these areas. He lead the PARADIGM compiler project for compiling programs for distributed memory multicomputers, the ProperCAD project for portable parallel VLSI CAD applications, the MATCH project on a MATLAB compilation environment for adaptive computing, and the PACT project on power aware compilation and architectural techniques. He is also the author of a book entitled "Parallel Algorithms for VLSI CAD" published by Prentice Hall, Inc., 1994. He has supervised 30 PhD and 35 MS student theses so far. Dr. Banerjee has received numerous awards and honors during his career. He received the IEEE Taylor L. Booth Education Award from the IEEE Computer Society in 2001. He became a Fellow of the ACM in 2000. He was the recipient of the 1996 Frederick Emmons Terman Award of ASEE's Electrical Engineering Division sponsored by Hewlett-Packard. He was elected to the fellow grade of IEEE in 1995. He received the University Scholar award from the University of Illinois for in 1993, the Senior Xerox Research Award in 1992, the IEEE Senior Membership in 1990, the US National Science Foundation's Presidential Young Investigators' Award in 1987, the IBM Young Faculty Development Award in 1986, and the President of India Gold Medal from the Indian Institute of Technology, Kharagpur, in 1981. The company that he has started called AccelChip has received the award for the "One of 50 emerging new technology companies in Illinois" in 2001. Prith has served as an Associate Editor of the *IEEE Transactions on Parallel and Distributed Systems*, the *IEEE Transactions on Computers*, the *Journal of Parallel and Distributed Computing*, the *IEEE Transactions on VLSI Systems*, and the *Journal of Circuits, Systems, and Computers*. He has served on the program and organizing committees of at least 30 conferences overview the past few years. He has also served as the Program Chair of the High-Performance Computing Conference in 1999, and the International Conference on Parallel Processing for 1995. He has served as General Chairman of the International Conference on Parallel and Distributed Computing Systems in 1997, and the International Workshop on Hardware Fault Tolerance in Multiprocessors 1989. He has been a consultant to many companies and was on the Technical Advisory Board of many companies such as Ambit Design Systems and Atrenta.