On reducing false sharing while improving locality on shared memory multiprocessors

Mahmut Kandemir* Alok Choudhary* J. Ramanujam† Prith Banerjee*

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. We present a mathematical framework that allows a clean description of the relationship between spatial locality and false sharing. We first show how to identify a severe form of multiple-writer false sharing and then demonstrate the importance of the interaction between optimization techniques aimed at enhancing locality and the techniques oriented toward reducing false sharing. Given the conflicting requirements, a compiler based approach to this problem holds promise. We investigate 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 Origin 2000 system, our approach brings an additional 9% improvement over a powerful locality optimization technique that uses both loop and data transformations. Also, our approach obtains an additional 19% improvement over an optimization technique that is oriented specifically toward reducing false sharing. Our study also reveals that in addition to reducing synchronization costs and improving memory subsystem performance, obtaining large granularity parallelism also helps these two optimization techniques, namely, enhancing locality and reducing false sharing, be compatible.

1 Introduction

With the increasing disparity between processor and memory speeds, exploiting the memory hierarchy characteristics of modern machines has become extremely important. In recent years, there has been much work on improving data locality of programs for uniprocessors [32, 24, 25, 26, 27, 23, 21, 16]. However, there is another critical issue to be considered in the case of shared-memory parallel machines, namely, *false sharing*. 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 [9, 13]. Here, 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 [9] show that a number of programs—that exhibit good spatial locality on uniprocessors—perform very poorly on multiprocessors.

The interaction between locality and false sharing on shared-memory parallel machines is quite well known. 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 [11]. 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.

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 as to which references may cause a *severe* form of *multiple-writer* false sharing if not addressed correctly. We then show how locality optimizations and loop 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 an analysis of this kind 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 Origin 2000 system demonstrate significant improvements. While the importance of outermost loop parallelism in handling other problems has been shown by previous researchers, 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 discusses preliminary experimental results. In Section 6, we discuss related work and conclude in Section 7 with a summary and a brief outline of ongoing and planned research.

2 Preliminaries

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 *spatial reuse* occurs. 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. 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

*Center for Parallel and Distributed Computing, Department of Electrical and Computer Engineering, Northwestern University, Evanston, IL 60208. e-mail: {mtk,choudhar,banerjee}@ece.nwu.edu

†Department of Electrical and Computer Engineering, Louisiana State University, Baton Rouge, LA 70803. e-mail: jxr@ee.lsu.edu

at least one of them *writes*) different data elements in the same coherence unit (cache line, memory page etc.) [13, 30, 9]. 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 $(x_1, ..., x_n)^T$ 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 [34]. Assuming a loop nest of depth n, an element p_i of the parallelism vector $\bar{p} = (p_1, ..., p_n)^T$ 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. Let $\bar{I}$ denote the *iteration vector* (consisting of loop indices starting from the outermost loop to the innermost). Under these assumptions, each array reference is represented as $\mathcal{L}\bar{I} + \bar{o}$, where the $m \times n$ matrix $\mathcal{L}$ is called the *access* (or *reference*) *matrix* [32] and the m-element vector $\bar{o}$ is referred to as the *offset vector*. The data reuse theory introduced by Wolf and Lam [32] and later refined by Li [24] can be used to identify the types of reuses 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 [32, 24].

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 non-zero 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. Figure 1 shows an example loop nest and illustrates these concepts. In this paper we focus on *self-reuses* (i.e., reuses that originate from individual references), but our approach can be extended to include *group-reuses* [32] 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 parallelization of a loop that carries spatial reuse [24]. For example, in Figure 2(a) 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 share (i.e., 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 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 non-zero entries in the false sharing vector $\bar{f}$ identify the references that can cause false sharing. Based on previous work in compilers, one can determine the desired values for $\bar{p}^T$, S' and $\bar{f}^T$. It is well known that a desired form of $\bar{p}^T$ has non-zero elements only at the beginning of it [34]. In effect, many compilers attempt to obtain a single 1 in the leftmost position which corresponds to parallelizing only the *outermost* loop in the nest. Previous work on optimizing locality [32, 25, 24] tells us that for each $\bar{s}' \in S'$, the index of the first non-zero 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., $\bar{s}' = (0, ..., 0, 1)^T$. 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. Also, 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 [6]). 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 [6]) 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. Note that although both reader-writer false sharing and multiple-writer false sharing can be avoided on machines that employ weak memory consistency models, it is always better for compiler to do it, since there is cache coherence overhead at runtime. Our approach can be extended to deal with reader-writer false sharing as well.

2.2 Examples

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 the code in Figure 2(a) and show how the false sharing vector is computed. Assume that the nest shown is

```
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
```

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

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

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

Figure 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; $\bar{s}'_u$ 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.

```
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 = 1, N
  do j = 1, N
    do k = 1, N
      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)

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

enclosed by a sequential timing loop and the i-loop is to be parallelized, i.e., $\bar{p} = (1,0,0)^T$. The spatial reuse summary vectors computed using reuse analysis are $\bar{s}'_u = (1,0,0)^T$, $\bar{s}'_v = (1,0,0)^T$, $\bar{s}'_w = (0,0,1)^T$. This means that the spatial reuses for 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, $\bar{f}^T = \bar{p}^T S' = (1,1,0)$. This means that if the i-loop is parallelized, then both $U(i,j)$ and $V(i,k)$ may incur multiple-writer false sharing. 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, 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 Figure 2(b). For the only reference shown, the spatial reuse vector is $\bar{s} = (0,1,-1)^T$ which leads to 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 either be $(1,0,0)^T$ or $(0,0,1)^T$ (assuming only one loop will be parallelized). To exploit outermost loop parallelism, it is better to select $\bar{p} = (1,0,0)^T$.

3 Impact of transformations on false sharing

Given a loop nest with a single LHS reference, the compiler faces the task of determining 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 $n \times n$ non-singular integer transformation matrix T. From data reuse theory [24], 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., [33, 25]), 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 Equation (1) we can check whether the reference incurs false sharing.

Locality-oriented approach: Using one of the algorithms in the literature (e.g., [24, 32]), 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 Equation (1) we can check whether the reference incurs false sharing.

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

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 an optimization technique such as data transformation (explained next) which affects only one of the two.

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 [27], [16], [23], [7], [14] 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 to explicitly-parallelized codes [7]. 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 [16] and utilized in this work, and then show the effect of data transformations on spatial locality and false sharing.

In our framework, we represent the memory layouts of multi-dimensional 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 called the *hyperplane constant* [28]. The hyperplane coefficients in Equation (2) are written as a hyperplane vector $\bar{g} = (g_1, g_2)^T$. A *hyperplane family* is a set of hyperplanes defined by $\bar{g}$ for different values of c.

A hyperplane family can be used to partially define the memory layout of a multi-dimensional array [16]. 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 is not important to us in this paper. Therefore, we can 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 anti-diagonal 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

$$(g_1, g_2)(\delta_1, \delta_2)^T = (g_1, g_2)(\delta'_1, \delta'_2)^T. \tag{3}$$

Consider an array stored in column-major order; i.e., the layout hyperplane vector is $(0, 1)^T$. Based on Equation (3), the array elements $(2, 3)^T$ and $(5, 3)^T$ belong to the same hyperplane. We say that two array elements that belong to the same hyperplane have *spatial locality* [16]. Although this definition of spatial locality is somewhat coarse, it is sufficient for the purposes of this work.

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 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.

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 Equation (3) given above, these two elements have spatial locality if $\bar{g}^T(\mathcal{L}\bar{I} + \bar{o}) = \bar{g}^T(\mathcal{L}\bar{I}' + \bar{o})$ (where $\bar{s}$ is the spatial reuse vector), or

$$\bar{g}^T\mathcal{L}(\bar{I}' - \bar{I}) = 0 \quad \text{or} \quad \bar{g}^T\mathcal{L}\bar{s} = 0 \quad \text{or} \quad \bar{g}^T \in Ker(\mathcal{L}\bar{s}). \tag{4}$$

Now, we have two important equations: Equation (1) and Equation (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., $\bar{p} = (1, 0, ..., 0)^T$. 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 $(1, 0, ..., 0)\bar{s}' = 0 \Rightarrow \bar{s}' = (0, \times, ..., \times, \times)^T$, where $\times$ stands for *dont-care*. A simple spatial reuse vector $\bar{s}$ that satisfies the $\bar{s}'$ vector above is $(0, ..., 0, 1)^T$. If we substitute this spatial reuse vector in Equation (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 [24] has observed that the form of the ideal spatial reuse vector is $(0, ..., 0, 1)^T$ 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.

Let us now consider the loop nest shown in Figure 2(c). 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.

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

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

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

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

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 Figure 2(d). Assume that we parallelize both the i and j loops. Thus, in mathematical terms, $\bar{p} = (1, 1, 0)^T$. 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,

$$\text{Array } U: \quad \bar{g}^T \in Ker\left\{(0,0,1)^T\right\} \Rightarrow G_u = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

$$\text{Array } V: \quad \bar{g}^T \in Ker\left\{(0,1,1)^T\right\} \Rightarrow G_v = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & -1 \end{pmatrix}$$

As in the previous example, with these layouts[1] 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 $(1, ..., 1, 0, ..., 0)^T$ can always be treated as $(1, 0, ..., 0)^T$; that is, all the outermost parallel loops can be collapsed into one loop. We can conclude:

In a given parallelism vector, if all the 1*s are in the leftmost positions consecutively (without a* 0 *in between them), then it is possible to reduce false sharing and optimize locality together for a given LHS reference.*

3.3 Outer and inner loop parallelization

It may not always be possible to obtain outermost loop parallelism in loop nests. For an n-nested loop, let D denote the dependence distance matrix [33], the columns of which are the (constant) dependence distance vectors [34]. 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 [33]. As noted earlier in this paper, in the case of an outermost parallel loop, we set the parallelism vector to $(1, 0, ..., 0)^T$ and then optimize each reference using $(0, 0, ..., 0, 1)^T$ 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 Figure 2(e). 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 can derive the code shown in Figure 2(f). 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 anti-diagonal. Now, we have spatial locality exploited in the innermost loop since successive iterations of the innermost loop access a given anti-diagonal; but, we incur false sharing at the coherence block boundaries. This example 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, $A_1, ..., A_\delta, A_{(\delta+1)}, ..., A_\gamma$. 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 Equation (4) for optimizing their locality. Specifically, let us consider an array j where $1 \leq j \leq \delta$. 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 k^{th} reference to this array and $\bar{s}_{jk}$ is the k^{th} spatial reuse vector. In the ideal case, we want to choose $\bar{s}_{jk} = (0, ..., 0, 1)^T$ for each reference k ($1 \leq k \leq t_j$). 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. 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 $\bar{s}_{jk} = (0, ..., 0, 1)^T$ for each reference k ($1 \leq k \leq t_j$).

(2) Sort the reference according to their *weights* in non-increasing 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 $\eta_j \leq t_j$ references optimized for the locality in the *innermost* loop. This process is independently repeated for all δ arrays in the first group.

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 $\delta + 1 \leq j \leq \gamma$. We divide the references for this array into two groups:

[1] G_u corresponds to a row-major layout, G_v represents a non-conventional layout. See [16] for a precise interpretation of layout matrices for arrays of dimension three and higher.

(1) the LHS references, $\mathcal{L}_{jk}$ where $1 \leq k \leq \Delta$, and

(2) the RHS references, $\mathcal{L}_{jk'}$ where $\Delta + 1 \leq k' \leq t_j$.

The constraints to be satisfied are 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 two options and select the one that performs better. In the first option, we set: $\bar{s}_{j1} = \bar{s}_{j2} = ... = \bar{s}_{j\Delta} \in Ker\{\bar{p}^T\}$ and $\bar{s}_{j(\Delta+1)} = \bar{s}_{j(\Delta+2)} = \bar{s}_{jt_j} = (0, ..., 0, 1)^T$. 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 $\bar{g}_j$. 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 $\bar{s}_{j1} = \bar{s}_{j2} = ... = \bar{s}_{j\Delta} = \bar{s}_{j(\Delta+1)} = \bar{s}_{j(\Delta+2)} = \bar{s}_{jt_j} = (0, ..., 0, 1)^T$. 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 the previous option.

After obtaining the solutions from these two options, we compare them 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 Figure 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. Our solution to this 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 is 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 which takes the layouts found in the most important nest into account. 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 [18, 17].

5 Experimental Results

We present preliminary experimental results obtained on an eight-processor SGI Origin 2000 distributed shared memory multiprocessor. This machine uses R10K processors each of which is a 4-way super-scalar 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 2-way set-associative on-chip data cache, and 4 MB external cache, which are called *primary* and *secondary* cache, respectively. The latency ratio between the first and second level caches is approximately $1 : 5$. The cache line size is 128 bytes

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 which 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 for the smallest weighted reference*
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* $1 \leq k \leq \Delta$
Set $\bar{s}_{jk'} = (0, ..., 0, 1)^T$ *for* $\Delta + 1 \leq k' \leq t_j$
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 for the smallest weight reference*
EndWhile
Compute cumulative weight*(Option 1) as the sum of the weights of the satisfied references*
//* Option 2 *//
Set $\bar{s}_{jk} = (0, ..., 0, 1)^T$ *for* $1 \leq k \leq \Delta$
Set $\bar{s}_{jk'} = (0, ..., 0, 1)^T$ *for* $\Delta + 1 \leq k' \leq t_j$
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 for the smallest weight reference*
EndWhile
Compute cumulative weight*(Option 2) as the sum of the weights of the satisfied references*
Compare the cumulative weights *of Options 1 and 2*
Choose the option with the larger weight
EndForeach
Using $\bar{g}_j^T$ *vectors found* $(1 \leq j \leq \gamma)$ *layout-transform the arrays in the nest*
End

Figure 3: An algorithm that reduces false sharing while improving locality.

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 twenty programs that can benefit from layout optimizations. For each code, we used an aggregate data size larger than the secondary cache capacity but smaller than the main memory. The second column of Table 2 gives the name of each code. The first nine programs are from benchmarks (B), programs 10 through 15 are codes from libraries (L), and programs 16 through 20 are example code fragments from an application called NWchem (A).

Table 1 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, `LOU` is the most powerful as it employs both loop and data transformations. 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 techniques as possible and selected the one that performs best. We also note that all arrays are padded [29] (where necessary) by a small amount to eliminate power-of-two sizes. In all these six versions, only *linear* loop and data transformations were used; tiling or loop unrolling were not used as they blur inherent locality. The `BAL` version refers to the approach discussed in this paper and `HND` is the hand-optimized version using both linear and non-linear (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 [22] may further improve the performance of the `HND` version. For all the versions, before transforming the code for locality and/or false sharing, we detected the largest granularity parallelism using the native compiler with locality optimizations turned off.[2] 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 the eight-processor case are presented in Table 2. In this table, the third column (`ORI`) gives the total execution times in *seconds*. Columns four through eleven 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`. The *imprv1* column gives the difference between `BAL` and the next best version from among the columns four through nine (usually `LOU`). This means that the `BAL` version outperforms a hypothetical approach which applies *all* these techniques and selects the best one. The *imprv2* column, on the other hand, shows the difference between `HND` and `BAL`.

From Table 2 we see that `FOU` is able to reduce the original execution times by nearly 9%. However, this is below the performance of `LOU` (which is around 19%). This is consistent with the conclusion of Torrellas et al. [30] that reducing false sharing at any cost is *not* a good idea. For lack of space, we did not present the results for just one processor.

The `BAL` version takes the parallelism decisions into account, and achieves a 28.49% 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% benefit, most of which can be obtained by applying a judicious tiling (tiling only the loops that carry some reuse [32]) after using `BAL`. In particular, the 14.14% performance gap between `BAL` and `HND` in application codes encourages us to use data-centric tiling [21] and control-centric tiling [12] 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.

[2] The locality optimizations are turned on for generating the `LOL` version.

6 Related work

Compiler researchers attacked the locality optimization problem from several points of view. Most of the research focused on enhancing the cache locality of scientific computations using loop transformations. Wolf and Lam [32] presented formal definitions of several types of reuse and offered a framework that uses unimodular loop transformations as well as tiling. Li [24] also focused on cache locality but considered general non-singular loop transformation matrices. McKinley et al. [26] presented a simple algorithm that unifies loop permutation, fusion, and distribution. Other researchers also considered tiling [12, 21, 32]. 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 the 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 [23], O'Boyle and Knijnenburg [27], and Kandemir et al. [16] 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 [7] and Kandemir et al. [17] 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 [20] 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. [30] 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 [13, 9] 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.

Eggers and Katz [10] and Bianchini and LeBlanc [2] also observe the impact of false sharing on parallel programs and propose

Table 1: 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, [24], [19]
LOD	locality optimized version using data (memory layout) transforms only	[23], [27], [16]
LOU	locality optimized version using both loop and data transforms	[18], [17], [7]
FOL	false sharing optimized version using loop (iteration space) transforms only	authors, [11], [3]
FOD	false sharing optimized version using data (memory layout) transforms only	authors, [30], [13]
FOU	false sharing optimized version using both loop and data transforms	authors, [8], [11]
BAL	version obtained using the approach discussed in this paper	authors
HND	hand optimized version using both linear and non-linear transforms	authors

Table 2: Results on 8 processors; **ave. (B)**, **(L)**, and **(A)** denote the averages for the benchmark (B), library (L), and application codes (A).

program #	code	ORI (sec.)	LOL (%)	LOD (%)	LOU (%)	FOL (%)	FOD (%)	FOU (%)	BAL (%)	HND (%)	*imprv1 (%)*	*imprv2 (%)*
1	hydro2d/T1	21.72	0.00	0.00	2.30	0.00	0.00	0.00	9.11	9.86	6.81	0.75
2	hydro2d/fct	9.61	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
3	vpenta	10.89	0.00	63.21	64.40	0.00	0.00	0.00	87.17	87.17	22.77	0.00
4	emit	3.27	2.90	5.05	7.05	4.90	6.33	6.90	30.01	30.01	22.96	0.00
5	btrix	20.19	13.07	55.91	70.04	19.16	−1.15	24.35	70.04	70.04	0.00	0.00
6	mxm	21.03	3.90	−16.51	4.40	8.21	8.21	8.21	9.87	24.55	1.66	14.68
7	cholsky	16.95	−8.21	0.00	0.00	−9.33	0.00	0.00	0.00	10.86	0.00	10.86
8	gmtry	13.90	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.08	0.00	1.08
9	adi	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	bakvec	10.27	2.05	4.20	15.65	4.49	6.91	6.91	40.86	40.86	25.21	0.00
11	htribk	9.25	−27.02	42.24	44.00	−11.00	−7.14	−7.14	44.00	51.07	0.00	7.07
12	qzhes	7.50	0.00	−1.00	0.00	0.00	8.95	0.00	8.95	14.69	0.00	5.74
13	fnorm	2.21	68.90	62.33	68.90	68.90	16.80	68.90	68.90	68.90	0.00	0.00
14	gfunp	17.28	0.00	0.00	10.11	0.00	0.00	0.00	20.04	22.67	9.93	2.63
15	r1mpyq	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	transpose	5.27	0.00	20.28	20.28	0.00	20.28	20.28	20.28	34.62	0.00	14.34
17	hnd_nw_hnd	9.17	2.30	3.15	3.15	2.30	3.15	3.15	3.15	5.55	0.00	1.40
18	hnd_nwhnd_tran	3.39	4.05	5.50	5.50	4.46	19.81	19.81	34.63	47.89	14.82	13.26
19	hnd_int_1e_studd	6.21	0.00	−9.50	0.00	13.21	13.21	13.21	29.85	49.55	16.69	19.70
20	hnd_whermt	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

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. [5] 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 [11] 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. [3, 4] 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. Cierniak and Li [8] 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 kernels that they used. Finally, Chow and Sarkar [6] proposed the modification of run-time scheduling parameters for eliminating multiple-writer false sharing. We believe that their solution is complementary to our approach.

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 presented a mathematical framework for studying the interaction between false sharing and locality for programs on shared-memory multiprocessors. We found that in those cases where the compiler can obtain outermost loop parallelism, it might be possible to simultaneously enhance spatial locality and reduce false sharing using memory layout transformations, which do not affect parallelism decisions already made by the compiler. On a collection of twenty programs drawn from various sources, the balanced approach presented in this paper brings about an additional 9% improvement over powerful loop and data transformations aimed specifically at locality, and shows a 19% improvement over techniques aimed specifically at reducing false sharing. This clearly demonstrates the benefits of balancing locality and false sharing. In those cases where the outermost loop cannot be executed in parallel, we need to decide whether to favor eliminating false sharing or favor optimizing locality; detailed profile informations might be useful in making this decision. We are currently working on embedding loop transformations—other than those aimed only at deriving parallel loops—into our framework. In addition, we are working on a formulation of the problem

as a set of inequality constraints that can then be solved by existing polyhedral tools.

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 NSF grants CCR-9357840 and CCR-9509143, and the Air Force Materials Command under contract F30602-97-C-0026. Prith Banerjee is supported in part by the DARPA under contract F30602-98-2-0144. J. Ramanujam is supported in part by an NSF Young Investigator Award CCR-9457768.

References

[1] J. Anderson, S. Amarasinghe, and M. Lam. Data and computation transformations for multiprocessors. In *Proc. 5th ACM SIGPLAN Symp. Prin. & Prac. Para. Prog.*, pp. 166–178, 1995.

[2] R. Bianchini and T. LeBlanc. Software caching on cache-coherent multiprocessors. In *Proc. 4th IEEE Symposium on Parallel and Distributed Processing*, December 1992.

[3] F. Bodin, E. Granston, and T. Montaut. Evaluating two loop transformations for reducing multiple-writer false sharing. In *Proc. 7th Workshop on Lang. & Compilers for Parallel Computing*, 1994.

[4] F. Bodin, E. Granston, and T. Montaut. Page-level affinity scheduling for eliminating false sharing. In *Proc. 5th Workshop on Compilers for Parallel Computing*, Malaga, Spain, 1995.

[5] W. Bolosky, R. Fitzgerald, and M. Scott. Simple but effective techniques for NUMA memory management. In *Proc. 12th ACM Symp. on Operating Systems Principles*, Dec 1989.

[6] J. Chow and V. Sarkar. False sharing elimination by selection of runtime scheduling parameters. In *Proc. 26th International Conference on Parallel Processing*, August 1997.

[7] M. Cierniak and W. Li. Unifying data and control transformations for distributed shared memory machines. In *Proc. SIGPLAN Conf. Prog. Lang. Des. Impl.*, pp. 205–217, 1995.

[8] M. Cierniak and W. Li. A practical approach to the compile-time elimination of false sharing for explicitly parallel programs. In *Proc. 10th Annual Intl. Conf. on High Perf. Comp.*, Canada, 1996.

[9] S. Eggers and T. Jeremiassen. Eliminating false sharing. In *Proc. International Conference on Parallel Processing*, volume I, pp. 377–381, August 1991.

[10] S. Eggers and R. Katz. The effect of sharing on the cache and bus performance of parallel programs. In *Proc. 3rd Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 257–270, April 1989.

[11] E. Granston and H. Wijshoff. Managing pages in shared virtual memory systems: getting the compiler into the game. In *Proc. International Conference on Supercomputing*, pp. 11–20, 1993.

[12] F. Irigoin and R. Triolet. Supernode partitioning. In *Proc. 15th Annual ACM Symp. on Principles of Programming Languages*, San Diego, CA, pp. 319–329, January 1988.

[13] T. Jeremiassen and S. Eggers. Reducing false sharing on shared memory multiprocessors through compile time data transformations. In *Proc. 5th ACM SIGPLAN Symp. Prin. & Prac. Par. Prog.*, 1995.

[14] M. T. Kandemir. *Compiler Techniques for Enhancing Data Locality.* Ph.D. Thesis, Department of Electrical Engineering and Computer Science, Syracuse University, Syracuse, NY, August 1999.

[15] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. A graph-based framework to detect optimal memory layouts for improving data locality. In *Proc. Intl. Para. Proc. Symp.*, April 1999.

[16] M. Kandemir, A. Choudhary, N. Shenoy, P. Banerjee, and J. Ramanujam, A linear algebra framework for automatic determination of optimal data layouts. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 10, No. 2, February 1999.

[17] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. Improving locality using loop and data transformations in an integrated framework. In *Proc. MICRO–31*, Dallas, TX, December 1998.

[18] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. A matrix-based approach to the global locality optimization problem. In *Proc. 1998 Intl. Conf. Parallel Arch. & Comp. Tech.,*, 1998.

[19] M. Kandemir, J. Ramanujam, A. Choudhary, and P. Banerjee. An iteration space transformation algorithm based on explicit data layout representation for optimizing locality. In *Proc. Workshop on Lang. & Comp. for Par. Comp.*, August 1998.

[20] K. Kennedy and K. S. McKinley. Optimizing for parallelism and data locality. In *Proc. 1992 ACM International Conference on Supercomputing (ICS'92)*, Washington, D.C., July 1992.

[21] I. Kodukula, N. Ahmed, and K. Pingali. Data-centric multi-level blocking. In *Proc. SIGPLAN '97 Conference on Programming Language Design and Implementation*, June 1997.

[22] M. Lam, E. Rothberg, and M. Wolf. The cache performance of blocked algorithms. In *Proc. 4th Int. Conf. Arch. Supp. for Prog. Lang. & Oper. Sys.*, April 1991.

[23] S.-T. Leung and J. Zahorjan. Optimizing data locality by array restructuring. Technical Report TR 95–09–01, Dept. of Computer Science and Engineering, University of Washington, September 1995.

[24] W. Li. *Compiling for NUMA Parallel Machines*. Ph.D. Thesis, Computer Science Department, Cornell University, Ithaca, New York, 1993.

[25] W. Li and K. Pingali. Access normalization: Loop restructuring for NUMA compilers. *ACM Transactions on Computer Systems*, (11)4:353–375, 1993.

[26] K. McKinley, S. Carr, and C.W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages & Systems*, 18(4):424–453, July 1996.

[27] M. O'Boyle and P. Knijnenburg. Non-singular data transformations: definition, validity, applications. In *Proc. 6th Workshop on Compilers for Parallel Computers*, pp. 287–297, Aachen, Germany, 1996.

[28] J. Ramanujam, and P. Sadayappan. Compile-time techniques for data distribution in distributed memory machines. In *IEEE Transactions on Parallel and Distributed Systems*, 2(4):472–482, Oct. 1991.

[29] G. Rivera and C.-W. Tseng. In *Proc. the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98)*, Montreal, Canada, June 1998.

[30] J. Torrellas, M. Lam, and J. Hennessey. False sharing and spatial locality in multiprocessor caches. *IEEE Transactions on Computers*, 43(6):651–663, June 1994.

[31] E. Torrie, C-.W. Tseng, M. Martonosi, and M. Hall. Evaluating the impact of advanced memory systems on compiler-parallelized codes. In *Proc. 1995 Intl. Conf. Parallel Arch. & Comp. Tech.,*, 1995.

[32] M. Wolf and M. Lam. A data locality optimizing algorithm. In *Proc. ACM SIGPLAN 91 Conf. Programming Language Design and Implementation*, pp. 30–44, June 1991.

[33] M. Wolf and M. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. on Parallel and Distributed Systems*, 2(4):452–471, 1991.

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