Page-level Affinity Scheduling for Eliminating False Sharing

Francois Bodin
Elana Granston
Thierry Montaut

CRPC-TR95532
May 1995

Center for Research on Parallel Computation
Rice University
6100 South Main Street
CRPC - MS 41
Houston, TX 77005

Page-level Affinity Scheduling for Eliminating False Sharing

François Bodin*

IRISA
Campus de Beaulieu
35042 Rennes, Cedex
France
bodin@irisa.fr

Elana D. Granston†

Rice University
Center for Research on Parallel Computation
6100 South Main Street
Houston, Texas 77005, USA
granston@cs.rice.edu

Thierry Montaut*

IRISA
Campus de Beaulieu
35042 Rennes, Cedex
France
montaut@irisa.fr

To appear in the *5th Workshop on Compilers for Parallel Computers*, Malaga, Spain, June 28-30, 1995

Abstract

To date, page management in distributed shared memory (DSM) systems has been primarily the responsibility of the run time system. However, there are some problems that are difficult to resolve efficiently at run time. Chief among these is false sharing. In this paper, we present a general technique for applying page-level affinity scheduling to eliminate false sharing resulting from regular references in numerical applications with loop-level parallelism. This a generalization of our previous work which was applied to one and two dimensional loops. We demonstrate the potential of these transformations experimentally.

1 Introduction

In large-scale multiprocessors, whether loosely or tightly coupled, there is typically some memory that is cheaper to access than other memory. Programming large scale multiprocessors directly (using message-passing primitives, for example) is not easy. Writing *good* compilers for these machines is not easy either. Typically compiler technology lags years behind architectural innovations.

Distributed shared memory (DSM) interfaces can potentially simplify the task of generating correct parallel code, by providing the compiler with the illusion of a global address space. This interface can be implemented in software or in hardware. DSM systems often maintain coherence automatically, using a unit of coherency that has the granularity of a page or cache line.

*Supported by the Esprit Agency DG XIII under Grant No. APPARC 6634 BRA III and Intel SSD under Grant No. 1 92 C 250 00 31318 01 2.

†Supported by a Postdoctoral Research Associateship in Computational Science and Engineering under National Science Foundation Grant No. CDA-9310307, and by the Center for Research on Parallel Computation under Grant No. CCR-9120008.

The success of the DSM abstraction depends heavily on page caching and the ability to exploit page-level locality. To date, proposed and implemented page-management strategies have relied heavily on the operating system, and secondarily on the hardware, to effect page placement and movement decisions. However, it is generally agreed upon that there are several problems that are difficult to handle if ignored until run time. Chief among these problems is false sharing, particularly multiple-writer false sharing, which arises when two or more processors are writing distinct data within the same coherency unit. The experimental results presented in Section 6 demonstrate the performance degradation that can be caused by false sharing.

False sharing is particularly problematic in software DSMs where coherency is maintained at the granularity of a page. Because software DSMs are typically implemented on top of or embedded into systems which were tuned for uniprocessor environments, page sizes are typically 4K or larger [LP92, KDCZ94].

In this paper, we present a technique for eliminating false sharing by applying page-level affinity scheduling. This technique is a generalization of our earlier work [Gra93, BGM94] which handled certain cases of one and two dimensional loops. The remainder of this paper is organized as follows. Section 2 presents an example of our technique. Section 3 presents the basic algorithm. Section 4 briefly addresses extensions for handling cases where multiple references which dictate conflicting iteration partitions. Section 5 addresses the the insertion and minimization of synchronization between loops. Section 6 presents experimental results. Section 7 discusses related research. Section 8 concludes this paper.

2 Applying Page-level Affinity Scheduling: An Example

Consider the following loop nest. Assume that the right hand side of the assignment statement does not cause any flow or anti dependences.

Loop Nest 1

```
        DO I1 = 0 TO N1-1
           DO I2 = 0 TO N2-1
R :           A[4*I1+3*I2+3] = h(I1,I2)
           END DO
        END DO
```

In this loop nest, the same page might be written during two or more distinct iterations. We can eliminate multiple writer false sharing resulting from $\mathcal{R}$ in two steps. First, we assign pages to processors in a block-cyclic fashion. Then, we effect a page-based owner-computes rule.

Assume that there are `P` processors $P_0, \ldots P_{P-1}$, that a page can hold exactly four elements of `A`, and that array `A` is page-aligned. Suppose that we partition pages into blocks of 2 consecutive pages and arbitrarily number the pages so that `A[0]` lies on page $\mathtt{K} = 0$. In this case, assigning pages to processors in a block-cyclic

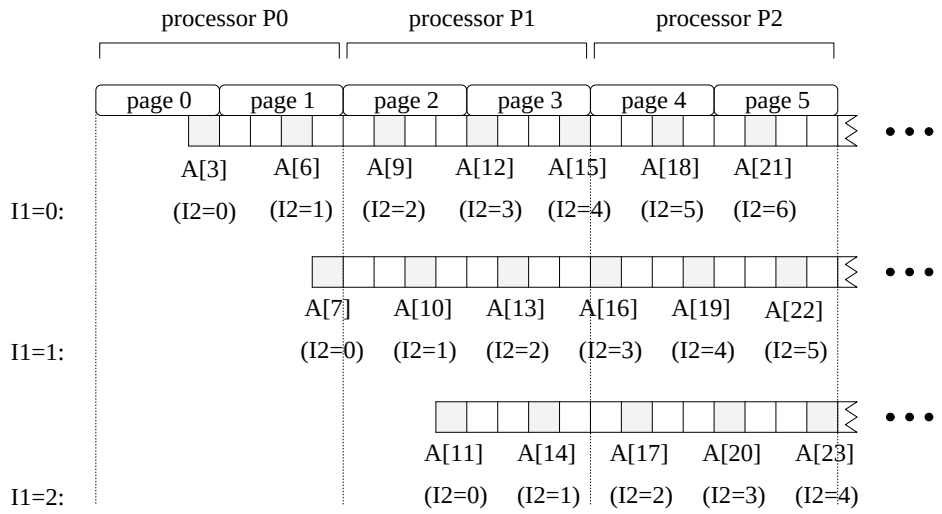


Figure 1: Example of applying page-level affinity scheduling to Loop Nest 1.

fashion causes processor *pid* to be assigned the set of pages

$$\rho(pid) = \{\ \mathtt{K} \mid \lfloor \mathtt{K}/2 \rfloor \bmod (\mathtt{P}) = pid\ \}\ .$$

Then we partition the iterations to ensure that processor *pid* is assigned all iterations where the element of referenced at $\mathcal{R}$ lies on a page in the set $\rho(pid)$. Figure 1 displays the resulting mapping between pages and processors as well as the iteration partitioning that is effected. False sharing is prevented by ensuring that no two processors write elements of `A` that lie on the same page. For example, consider array elements `A[12:15]` which all lie on page $\mathtt{K} = 3$ are accessed during iterations ($\mathtt{I_1}$,$\mathtt{I_2}$) = (0,3), (1,2), (2,1), and (0,4), respectively. Because page 3 lies in $\rho(P_1)$, we partition iterations so that all four of these iterations are assigned to processor P_1.

The constraints that must be met to effect page-based owner computes rule can be expressed by the following series of inequalities:

$$\begin{cases} 0 \le \mathtt{I_1} \le \mathtt{N_1} - 1 \\ 0 \le \mathtt{I_2} \le \mathtt{N_2} - 1 \\ 4 * \mathtt{K} \le 4 * \mathtt{I_1} + 3 * \mathtt{I_2} + 3 \le 4 * (\mathtt{K} + 1) - 1 \\ 2 * (\mathtt{n} * \mathtt{P} + pid) \le \mathtt{K} \le 2 * (\mathtt{n} * \mathtt{P} + pid + 1) - 1 \\ 0 \le \mathtt{n} \le \left\lceil \frac{sizeof(\mathtt{A})}{8 * \mathtt{P}} \right\rceil - 1 \end{cases}$$

We can solve successively for $\mathtt{I_2}$, $\mathtt{I_1}$, $\mathtt{K}$, and $\mathtt{n}$ to yield:

$$\left\{\begin{array}{rcccl}
\underbrace{\max\left(0, \left\lceil \frac{4*\mathtt{K}-4*\mathtt{I_1}-3}{3} \right\rceil\right)}_{I_2min} & \leq & \mathtt{I_2} & \leq & \underbrace{\min\left(\mathtt{N_2}-1, \left\lfloor \frac{4*\mathtt{K}-4*\mathtt{I_1}}{3} \right\rfloor\right)}_{I_2max} \\
\underbrace{\max\left(0, \left\lceil \frac{4*\mathtt{K}-3*\mathtt{N_2}}{4} \right\rceil\right)}_{I_1min} & \leq & \mathtt{I_1} & \leq & \underbrace{\min(\mathtt{N_1}-1, \mathtt{K})}_{I_1max} \\
\underbrace{\max(0, 2*\mathtt{n}*\mathtt{P}+2*pid)}_{Kmin} & \leq & \mathtt{K} & \leq & \underbrace{\min\left(2*\mathtt{n}*\mathtt{P}+2*pid+1, \left\lfloor \frac{4*\mathtt{N_1}+3*\mathtt{N_2}-4}{4} \right\rfloor\right)}_{Kmax} \\
\underbrace{0}_{nmin} & \leq & \mathtt{n} & \leq & \underbrace{\min\left(\left\lceil \frac{sizeof(\mathtt{A})}{8*\mathtt{P}} \right\rceil - 1, \left\lfloor \frac{4*\mathtt{N_1}+3*\mathtt{N_2}-4-8*pid}{8*\mathtt{P}} \right\rfloor\right)}_{nmax}
\end{array}\right.$$

From these inequalities, we can directly generate the SPMD node code that effects our partitioning:

Loop Nest 2

$\mathcal{R}$:

```
pid = getmypid()
...
DO n= nmin TO nmax
   DO K= Kmin TO Kmax
      DO I1 = I1min TO I1max
         DO I2 = I2min TO I2max
            A[4*I1+3*I2+3] = h(I1,I2)
         END DO
      END DO
   END DO
END DO
```

In the transformed loop nest, each processor *pid* only executes iterations where data is written to pages it owns. The `n` loop iterates over blocks owned by *pid* that are accessed during a given iteration of the original $\mathtt{I_1}$ loop. The `K` loop iterates over pages within a given block. All writes to a given page are performed by a single processor within a single iteration of the `K` loop. Thus multiple-writer false sharing is eliminated.

By using the same mapping of pages to processors when applying this transformation to other loop nests with references to array `A`, we achieve page-level affinity scheduling.

Note that the original loop nest (Loop Nest 1) has an output data dependence that is carried by the outer loop. Our affinity-scheduling technique preserves this dependence in the transformed loop nest, by ensuring that all writes of elements of array `A` to any given page are performed by the same processor in their original order.

3 Algorithm Overview

Consider the following generic loop nest. Assume that $f_1, \ldots, f_D$ are affine functions of the loop index variables.

Loop Nest 3

```
        DO I1 = L1 TO U1
           DO I2 = L2 TO U2
              ...
              DO IJ = LJ TO UJ
                 ...
                 DO IM = LM TO UM
R :                 A[f1(I1,...,IM), f2(I1,...,IM),..., fD(I1,...,IM)] = h(I1,...,IM)
                 END DO
                 ...
              END DO
              ...
           END DO
        END DO
```

Page-level affinity scheduling is implemented by generating affine inequalities that represent the constraints on the original loop iteration variables and for two new loop iteration variables n and K. In this section, we first show how to generate the system of inequalities. Then we discuss the issues of solving the system of inequalities using the Fourier-Motzkin algorithm. We then consider the order for performing the pair-wise elimination.

3.1 Setting Up the Constraints

First, we generate the constraints that represent the range for each loop index variable $\mathtt{I_J}$, $0 \leq \mathtt{J} \leq \mathtt{I_M}$:

$$\mathtt{L_J} \leq \mathtt{I_J} \leq \mathtt{U_J} \ .$$

Suppose that a page contains precisely m elements of A. Let $o(\mathtt{A})$ be the offset of A[0] on some page. If $o(\mathtt{A})$ is page-aligned, then $o(\mathtt{A})$=0. Let A[$f(\mathtt{I_1},\mathtt{I_2},\ldots,\mathtt{I_M})$] be the linearized version of $\mathcal{R}$. Then, $\mathtt{A}[m * \mathtt{K} - o(\mathtt{A})]$ is the first element on page K, and $\mathtt{A}[m * (\mathtt{K}+1) - o(\mathtt{A}) - 1]$ is the last. We add in the constraint that specifies the elements of A that lie on page K:

$$m * \mathtt{K} - o(\mathtt{A}) \leq f(\mathtt{I_1}, \mathtt{I_2}, \ldots, \mathtt{I_M}) \leq m * (\mathtt{K}+1) - o(\mathtt{A}) - 1 \ .$$

Pages are assigned to processors in blocks of b pages in a block-cyclic fashion. Assuming that the blocks assigned to each processor are numbered from 0, the n'th block assigned to processor pid contains pages $\mathtt{K} = b * (\mathtt{n} * \mathtt{P} + pid)$ through $\mathtt{K} = b * (\mathtt{n} * \mathtt{P} + pid + 1) - 1$. This yields the constraint

$$b * (\mathtt{n} * \mathtt{P} + pid) \leq \mathtt{K} \leq b * (\mathtt{n} * \mathtt{P} + pid + 1) - 1 \ .$$

Finally, we must generate constraint on $\mathtt{n}$. There are $\left\lceil \frac{sizeof(\mathtt{A}) + o(\mathtt{A})}{b * m} \right\rceil$ blocks of pages. Assuming that there are $\mathtt{P}$ processors, each processor is assigned at most $\left\lceil \frac{sizeof(\mathtt{A}) + o(\mathtt{A})}{b * m * \mathtt{P}} \right\rceil$ blocks. Therefore,

$$0 \leq \mathtt{n} \leq \left\lceil \frac{sizeof(\mathtt{A}) + o(\mathtt{A})}{b * m * \mathtt{P}} \right\rceil - 1 \ .$$

This gives us the following inequality system:

$$\begin{cases} \mathtt{L_1} \leq & \mathtt{I_1} & \leq \mathtt{U_1} \\ \ldots \leq & \ldots & \leq \ldots \\ \mathtt{L_M} \leq & \mathtt{I_M} & \leq \mathtt{U_M} \\ b * (\mathtt{n} * \mathtt{P} + pid) \leq & \mathtt{K} & \leq b * (\mathtt{n} * \mathtt{P} + pid + 1) - 1 \\ 0 \leq & \mathtt{n} & \leq \left\lceil \frac{sizeof(\mathtt{A}) + o(\mathtt{A})}{b * m * \mathtt{P}} \right\rceil - 1 \\ m * \mathtt{K} - o(\mathtt{A}) \leq & f(\mathtt{I_1}, \mathtt{I_2}, \ldots, \mathtt{I_M}) & \leq m * (\mathtt{K} + 1) - o(\mathtt{A}) - 1 \end{cases}$$

3.2 Solving Systems of Integer Inequalities

We currently solve the system of inequalities using Fourier-Motzkin elimination [Sch86]. In general, given a matrix A and a vector b, standard Fourier-Motzkin finds a **rational** solution to $Ax \leq b$, if one exists. Standard Fourier-Motzkin consists of rewriting the system of inequalities as follows[1]:

$$(S) \begin{cases} \mathtt{I_1} + a_i x' \leq \beta_i & i = 1, ..., l' \\ -\mathtt{I_1} + a_i x' \leq \beta_i & i = l' + 1, ..., l'' \\ a_i x' \leq \beta_i & i = l'' + 1, ..., l \end{cases}$$

where $x' = (\mathtt{I_2}, ..., \mathtt{I_M})$ and $a_1, ..., a_l$ are rows of the matrix A.
From the first two lines of (S) we get

$$max_{l'+1 \leq j \leq l''}(a_j x' - \beta_j) \leq \mathtt{I_1} \leq min_{1 \leq i \leq l'}(\beta_i - a_i x')$$

[1] Individual inequalities may have been multiplied by positive scalars.

and a new system (S') by eliminating $\mathtt{I_1}$ from (S):

$$(S') \begin{cases} (a_i + a_j)x' \leq \beta_i + \beta_j & i = 1, ..., l',\ j = l' + 1, ..., l'' \\ a_i x' \leq \beta_i & i = l'' + 1, ..., l \end{cases}$$

By repeating this procedure we can successively eliminate the first M-1 components of vector x, resulting in a trivial system with one unknown.

Relying on Fourier Motzkin elimination has several potential drawbacks. First, the complexity of Fourier-Motzkin elimination itself is exponential in the worst case, and too expensive to be generally relied upon in a commercial compiler. Second, loop iteration spaces are not necessarily convex integer subspaces while Fourier-Motzkin elimination assumes convex rational subspaces[2]. The consequence of this is that the resulting code may contain *empty iterations*, namely for one or more iterations of an outer loop, an inner loop has zero trips.

Ancourt and Irigoin [AI91] and Le Fur [Fur95] have focused on addressing the first concern by developing techniques for eliminating redundant constraints, key source of inefficiency in the basic algorithm. Our approach is to precompute solutions for commonly occurring cases. An example of precomputation is given in Figure 2[3]. For the exceptional cases where we must apply Fourier-Motzkin at compile time, however, we can exploit their optimizations for eliminating redundant constraints.

The second concern, the "empty iteration" problem, appears to be more of a problem in theory than in practice. Empty iterations arise primarily from iterating over pages which do not contain any accessed data. In particular, the n and K loops iterate over the smallest consecutive set of pages that contain the elements of A to be accessed within enclosing loops. If any of these pages are empty, there will be an empty iteration. Empty iterations do not cause incorrect results. They simply add loop overhead. Because pages are so large, the number of empty pages is likely to be very small. Moreover, the extra loop overhead per empty iteration is also very small, especially after applying standard optimizations to the loop bound expressions. For completeness sake, however, we note that there are several techniques for eliminating them should they become problematic [Pug91, Mon95].

An alternative to Fourier-Motzkin has been proposed by Feautrier [Fea89]. He handles systems of linear inequalities using the simplex method [Sch86]. He proposes a parametrized simplex algorithm (PIP) which can be used to compute the loop bounds. The advantage of the simplex method is that it does not introduce redundant constraints in the system. However the algorithm has to be applied for each loop bound and a simplification of the loop bounds must be performed before the results can be used [JC93].

[2] The lack of convexity arises from the integer division.

[3] $sign(t) = 1$ if $t > 0$, -1 if $t < 0$, and 0 otherwise. $t^+ = \max(t, 0)$. $t^- = \max(-t, 0)$.

```
      DO I1=0 TO N1 − 1
         DO I2=0 TO N2 − 1
R  :        A[c1 ∗ I1 + c2 ∗ I2 + λ] = ...
         END DO
      END DO
```

$$\begin{cases} 0 \leq & \mathtt{I_1} & \leq \mathtt{N_1} - 1 \\ 0 \leq & \mathtt{I_2} & \leq \mathtt{N_2} - 1 \\ m * \mathtt{K} - o(\mathtt{A}) \leq & c_1 * \mathtt{I_1} + c_2 * \mathtt{I_2} + \lambda & \leq m * (\mathtt{K}+1) - o(\mathtt{A}) - 1 \\ b * (\mathtt{n} * \mathtt{P} + pid) \leq & \mathtt{K} & \leq b * (\mathtt{n} * \mathtt{P} + pid + 1) - 1 \\ 0 \leq & \mathtt{n} & \leq \left\lceil \frac{sizeof(\mathtt{A}) + o(\mathtt{A})}{b * m * \mathtt{P}} \right\rceil - 1 \end{cases}$$

1. $\mathtt{I_2}$ elimination yields

$$\begin{array}{rcl} 0 \leq & \mathtt{I_2} & \leq \mathtt{N_2} - 1 \\ \left\lceil \frac{m * \mathtt{K} - o(\mathtt{A}) - c_1 * \mathtt{I_1} - \lambda + sign^-(c_2)(m-1)}{c_2} \right\rceil \leq & \mathtt{I_2} & \leq \left\lfloor \frac{m * \mathtt{K} - o(\mathtt{A}) - c_1 * \mathtt{I_1} - \lambda + sign^+(c_2)(m-1)}{c_2} \right\rfloor \end{array}$$

2. $\mathtt{I_1}$ elimination yields

$$\begin{array}{rcl} 0 \leq & \mathtt{I_1} & \leq \mathtt{N_1} - 1 \\ \left\lceil \frac{m * \mathtt{K} - o(\mathtt{A}) - \lambda + sign^-(c_1)(m-1) - e_1}{c_1} \right\rceil \leq & \mathtt{I_1} & \leq \left\lfloor \frac{m * \mathtt{K} - o(\mathtt{A}) - \lambda + sign^+(c_1)(m-1) - e_2}{c_1} \right\rfloor \end{array}$$

where

$$\begin{array}{rl} e_1 = & sign^+(c_1c_2)c_2(\mathtt{N_2} - 1) \\ e_2 = & sign^-(c_1c_2)c_2(\mathtt{N_2} - 1) \end{array}$$

3. $\mathtt{K}$ elimination yields

$$\begin{array}{rcl} b * (\mathtt{n} * \mathtt{P} + pid) \leq & \mathtt{K} & \leq b * (\mathtt{n} * \mathtt{P} + pid + 1) - 1 \\ \left\lceil \frac{sign^-(c_1)(-c_1 * (\mathtt{N_1} - 1) - e_3) + sign^+(c_1)(-e_4)}{m} \right\rceil \leq & \mathtt{K} & \leq \left\lfloor \frac{sign^-(c_1)(-e_4) + sign^+(c_1)(c_1 * (\mathtt{N_1} - 1) - e_3)}{m} \right\rfloor \end{array}$$

where

$$\begin{array}{rl} e_3 = & -o(\mathtt{A}) - \lambda + sign^-(c_1)(m-1) - sign^+(c_1c_2)c_2(\mathtt{N_2} - 1) \\ e_4 = & -o(\mathtt{A}) - \lambda + sign^+(c_1)(m-1) - sign^-(c_1c_2)c_2(\mathtt{N_2} - 1) \end{array}$$

4. $\mathtt{n}$ is kept unchanged.

Figure 2: Precomputed loop bounds for two-dimensional loop nests without flow dependences or anti dependences. $c_1, c_2 \neq 0$.

3.3 Determining the Elimination Order

The next step is to determine the order of elimination. We would like to solve the inequalities in the order that will maximize data locality subject to dependence constraints. We assume that standard loop level transformations for increasing parallelism, temporal locality and spatial locality [BEJW92, KM92, WL91] have already been applied. Therefore, we assume the loops $\mathtt{I}_1, \ldots, \mathtt{I}_\mathtt{M}$ must remain in the same order with respect to each other.

The simplest case arises when the loop nest contains no flow or anti-dependence. Recall from Section 2, that our transformation automatically preserves output dependences where the source and sink are the same, because it preserves the order of writes to any given page. In this case, we can apply Fourier-Motzkin elimination in the order $\mathtt{I}_1, \mathtt{I}_2, \ldots, \mathtt{I}_\mathtt{M}, \mathtt{K}, \mathtt{n}$ as in our earlier example. In general, synchronization is needed around the outermost loop. In some cases, however, our transformation exposes opportunities to eliminate this synchronization (Section 5).

When the loop nest does contain flow or anti-dependences, the above elimination order may not be legal. This problem is similar to that which arises when compiling languages such as HPF [KLS+94, BCZ92, AFMP95, FSHK+91]. Suppose that innermost loop that carries a flow or input dependence is the $\mathtt{I}_\mathtt{J}$ loop. Then we can apply page level affinity scheduling by replicating loops $\mathtt{I}_1, \ldots, \mathtt{I}_\mathtt{J}$ (i.e., each processor executes all iterations of these loops) and partitioning loops $\mathtt{I}_\mathtt{J} + 1, \ldots, \mathtt{I}_\mathtt{M}$. This results in the following set of constraints on variables $\mathtt{I}_\mathtt{J} + 1, \ldots, \mathtt{I}_\mathtt{M}$:

$$\begin{cases} \mathtt{L}_{\mathtt{J}+1} \leq & \mathtt{I}_{\mathtt{J}+1} & \leq \mathtt{U}_{\mathtt{J}+1} \\ \ldots \leq & \ldots & \leq \ldots \\ \mathtt{L}_\mathtt{M} \leq & \mathtt{I}_\mathtt{M} & \leq \mathtt{U}_\mathtt{M} \\ b * (\mathtt{n} * \mathtt{P} + pid) \leq & \mathtt{K} & \leq b * (\mathtt{n} * \mathtt{P} + pid + 1) - 1 \\ 0 \leq & \mathtt{n} & \leq \left\lceil \frac{sizeof(\mathtt{A}) + o(\mathtt{A})}{b * m * \mathtt{P}} \right\rceil - 1 \\ m * \mathtt{K} - o(\mathtt{A}) \leq & f(\mathtt{I}_1, \mathtt{I}_2, \ldots, \mathtt{I}_\mathtt{M}) & \leq m * (\mathtt{K} + 1) - o(\mathtt{A}) - 1 \end{cases}$$

We then eliminate variables in the order $\mathtt{I}_\mathtt{M}, \ldots, \mathtt{I}_{\mathtt{J}+1}, \mathtt{K}, \mathtt{n}$. In general, synchronization must be inserted around the $\mathtt{n}$ loop. Exceptions are discussed in Section 5.

We note that false sharing elimination may conflict with the parallelization process. In particular, parallelism is maximized by moving parallel loops outermost. The consequence of this is that iterations of innermost loops which are sequentialized due to flow or anti-dependences must then be treated indivisibly, unless we are willing to insert synchronization into innermost loops. Treating iterations of innermost loops indivisibly precludes the application of our page-based owner computes rule. Inserting synchronization into innermost loops effectively involves trading false sharing for "true sharing" and would likely outweigh the performance benefits of applying our scheduling strategy. Hence this option was not considered. Alternate

approaches to eliminating false sharing from loops with innermost sequential loops, for example by modifying data layout, is addressed in Montaut [Mon95].

4 Optimizing Loop Nests with Multiple Write References

In practice, programs often contain loop nests with more than one static write reference. This section describes techniques for applying page-level affinity scheduling to loops with multiple write references. Before the techniques can be described, the notions of *array group* and *reference group* are needed.

An *array group* is a set of arrays which have the same page alignment, the same array dimensions (excluding the outermost dimension) and the same size elements. For example, in Loop Nest 4, `ZU` and `ZV` have the same size elements. If they also have the same innermost dimension and same offset, then they belong to the same array group. Otherwise, they belong to distinct array groups.

A *reference group* includes static write references with two characteristics. First, the arrays being referenced must belong to the same array group. Second, the subscript expressions of the arrays being referenced at these points must be the same. For example, in Loop Nest 4, the write references to `ZU` and `ZV` have the same subscript expression. If both arrays belong to the same array group, then both of these write references belong to the same reference group. Otherwise, they belong to distinct reference groups. In general, there are at most a few reference groups within a given loop nest. This is especially true if arrays are aligned with page boundaries.

A reference group has the desirable property that the footprint of each reference in the group moves through memory at the same speed and crosses page boundaries at the same time. Consequently, all references in within a single reference group have the same sharing pattern. If we can partition iterations to eliminate multiple-writer false sharing for one reference in a group G, we have eliminated it for all references *within* G. If there is only one reference group, then multiple-writer false sharing is eliminated altogether.

Suppose that there is more than one reference group. If loop distribution is legal, we can distribute the loops so that each new loop nest only contains write references from a single reference group. Then we can handle each loop nest independently. By default, a synchronization instruction is required between the two loop nests. Exceptions are discussed in Section 5.

When loop distribution is not feasible, we can eliminate page-level sharing within some reference group. For the remainder, we can reduce the number of page migrations that page-level sharing causes by reducing ping-pong effects (i.e., the repetitive bouncing of a falsely shared page between processors) using a loop transformation that we developed specifically for this purpose [BGM94]. Depending on the coherency protocol, most of the page migrations attributed to false sharing may actually be caused by ping-pong effects. Consequently, we have found that this hybrid strategy works well in practice [BGM94].

Loop Nest 4

```
      /* Key loop nest from Lawrence Livermore Kernel 18 */
        DO I1 = 2 TO 6
          DO I2 = 2 TO N-1
R^ZU:       ZU[I2,I1] = ZU[I2,I1] + S * (ZA[I2,I1] * (ZZ[I2,I1] - ZZ[I2+1,I1]))
                                  - ZA[I2-1,I1] * (ZZ[I2,I1] - ZZ[ I2-1,I1])
                                  - ZB[I2,I1] * (ZZ[I2,I1] - ZZ[I2,I1-1])
                                  + ZB[I2,I1+1] * (ZZ[I2,I1) - ZZ[ I2,I1+1)))

R^ZV:       ZV[I2,I1] = ZV[I2,I1] + S * (ZA[I2,I1] * (ZR[I2,I1] - ZR[I2+1,I1])
                                  - ZA[I2-1,I1] * (ZR[I2,I1] - ZR[ I2-1,I1])
                                  - ZB[I2,I1+1] * (ZR[I2,I1] - ZR[ I2,I1-1])
                                  + ZB[I2,I1+1] * (ZR[I2,I1] - ZR[ I2,I1+1]))
          END DO
        END DO
```

5 Synchronizing Between Loop Nests

In the following section, we discuss the insertion and optimization of barrier synchronization, the only synchronization primitive considered in this paper. In general, whenever a a pair of loop nests have a flow, anti or output dependence between them (i.e., the source of the dependence is in one loop nest and the sink is in the other), a barrier must be inserted between the loop nests to enforce the dependence. This type of dependence is known as a *cross-loop dependence*. For efficiency, however, we wish to minimize the number of barriers that we insert. To do this, we rely on the technique presented in O'Boyle et al. [OKB]. We describe this technique briefly here.

By using the same mapping between pages and processors across loops, our transformation to eliminate false sharing achieves page-level affinity scheduling. This is equivalent to standard data distribution, such as the one in HPF, with the major difference that only one data distribution at the page level is available. In many cases, this scheduling strategy has the side effect of ensuring that both the source and sink of cross-loop dependence will be executed by the same processor. In these cases, a barrier is not needed to enforce the dependence. There are three such cases. We describe them in the next three subsections. Then we present a brief example.

5.1 Case 1: Flow Dependence Exceptions

Consider the two loop nests below with a flow dependence from $\mathcal{R}_1$ to $\mathcal{R}_3$. Assume that f_1, f_2 and f_3 are affine functions of loop index variables.

Loop Nest 5

```
          DO I1 = L1^I TO U1^I
             DO I2 = L2^I TO U2^I
R1:             A[f1(I1,I2)] = ···
             END DO
          END DO
          ···
          DO J1 = L1^J TO U1^J
             DO J2 = L2^J TO U2^J
R2:             B[f2(J1,J2)] = ···
R3:             ···          = A[f3(J1,J2)]
             END DO
          END DO
```

A barrier must be inserted between the two loop nests to enforce this flow dependence unless all three of the following conditions are met:

- $f_2 = f_3$.
- Arrays `A` and `B` belong to the same array group.
- Page-level affinity scheduling is applied to both loop nests based on $\mathcal{R}_1$ and $\mathcal{R}_2$, respectively.

5.2 Case 2: Anti Dependence Exceptions

Consider the two loop nests below with a anti dependence from $\mathcal{R}_2$ to $\mathcal{R}_3$.

Loop Nest 6

```
          DO I1 = L1^I TO U1^I
             DO I2 = L2^I TO U2^I
R1:             B[f1(I1,I2)] = ···
R2:             ···          = A[f2(I1,I2)]
             END DO
          END DO
          ···
          DO J1 = L1^J TO U1^J
             DO J2 = L2^J TO U2^J
R3:             A[f3(J1,J2)] = ···
             END DO
          END DO
```

A barrier must be inserted between the two loop nests to enforce this anti dependence unless all three of the following conditions are met:

- $f_1 = f_2$.
- Arrays `A` and `B` belong to the same array group.
- Page-level affinity scheduling is applied to both loop nests based on $\mathcal{R}_1$ and $\mathcal{R}_3$, respectively.

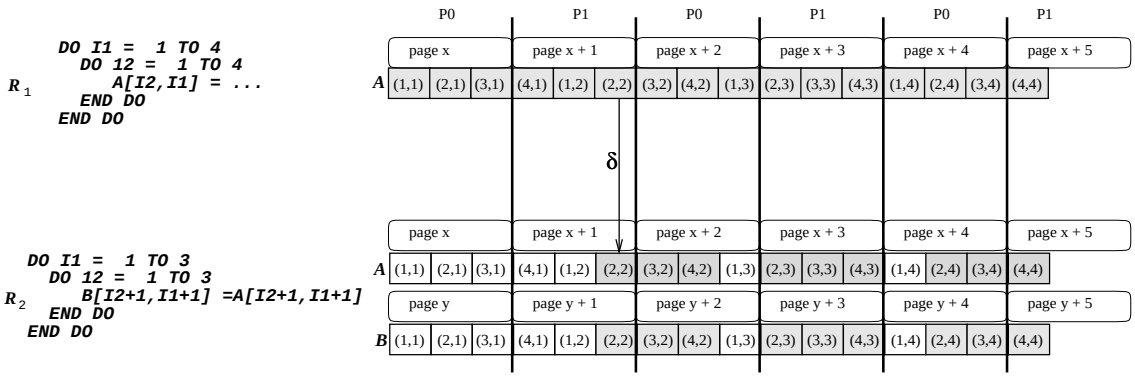


Figure 3: Example with two processors. Page-level affinity scheduling eliminates false sharing and ensures that the source and sink of the flow dependence are both executed on the same processor. Consequently, synchronization between the loops is not needed. ($b = 1,\ m = 3$)

5.3 Case 3: Output Dependence Exceptions

Consider the two loop nests below with an output dependence from $\mathcal{R}_1$ to $\mathcal{R}_2$.

Loop Nest 7

```
        DO I1 = L1^I TO U1^I
           DO I2 = L2^I TO U2^I
R1:           A[f1(I1,I2)] = ...
           END DO
        END DO
        ...
        DO J1 = L1^J TO U1^J
           DO J2 = L2^J TO U2^J
R2:           A[f2(J1,J2)] = ...
           END DO
        END DO
```

No synchronization is needed to enforce this output dependence if page-level affinity scheduling is applied to both, based on $\mathcal{R}_1$ and $\mathcal{R}_2$, respectively.

5.4 Example

Consider the loop nest in Figure 3 (left). There is a cross-loop flow dependence (δ) between references $\mathcal{R}_1$ and $\mathcal{R}_2$. Suppose that we partition the iterations of each loop nest in a traditional manner, for example, by assigning consecutive chunks of $N/\mathtt{P}$ iterations to each processor, where N is the problem size and $\mathtt{P}$ is the number of processors. Then a barrier is needed between the two loops. Suppose instead that we apply page-level affinity scheduling to each loop nest based on the write references $\mathcal{R}_1$ and $\mathcal{R}_2$, respectively. The result is depicted in Figure 3 (right). Because this dependence falls under Case 1, the barrier is no longer needed. Both the source and sink of the dependence are now executed on the same processor, so the dependence is enforced without synchronization.

6 Experimental Results

Page-level affinity scheduling (PLAS), except for the precomputation, has been implemented in the Fortran-S compiler [BKP93]. The Fortran-S compiler generates SPMD-style code that runs on the iPSC/2 under the KOAN software DSM [LP92]. The KOAN DSM system is embedded in the operating system of the iPSC/2. Pages of size 4 KB are physically distributed across processors' local memories. KOAN uses a distributed-manager algorithm based on [Li86], with an invalidation protocol that ensures that the shared memory is coherent at all times [CF78]. Under this protocol, pages can have one of three access modes: *read-only*, *write-exclusive* and *invalid*. Multiple copies of a page are permitted only when all copies are in read-only mode. When a processor needs to write to a page and either has a read-only copy or no copy at all, the processor must send a message to the page's manager requesting write-exclusive access. Once all other copies of that page are invalidated, a write-exclusive copy is sent to the requesting processor, which can then proceed with its write.

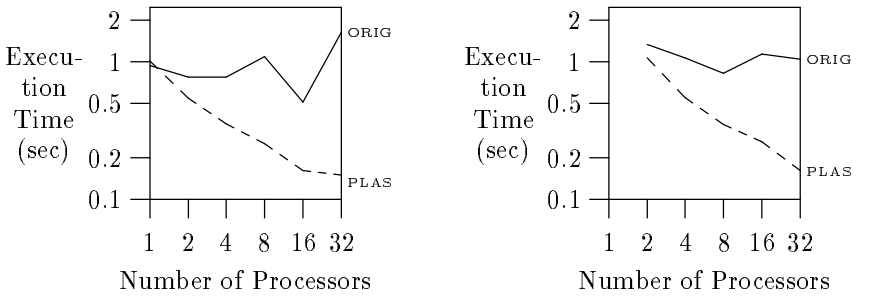
Several Fortran-77 benchmarks were studied. First, each benchmark was manually optimized to exploit both parallelism and locality, using stripmining and interchanging where appropriate. Then the benchmarks were then input into the Fortran-S compiler.

Two versions of each benchmark were generated: ORIG and PLAS. In the ORIG version, the outermost parallel loop was partitioned across processors so that each processor was assigned a consecutive chunk of $N/\mathtt{P}$ iterations, where N was the total number of iterations in the loop. The first chunk of $N/\mathtt{P}$ iterations was assigned to Processor P_0, the second chunk to Processor P_1, and so forth. Using the same mapping of chunks to processors for all parallel loops naturally afforded some processor affinity for some of the benchmarks. Barrier synchronization was inserted around parallel loops.

For the PLAS version, each processor was assigned chunk of b "pages" worth of iterations, where b that was chosen to yield the partitioning that would be closest to $N/\mathtt{P}$ to minimize the differences between the two versions. Barriers were inserted when necessary.

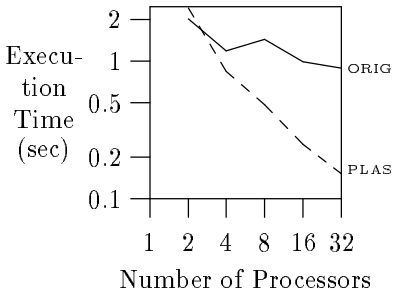
6.1 DMXPY

Loop Nest 8 depicts the Fortran kernel DMXPY from LINPACKD [DBMS79] which performs matrix–vector multiplication. For this experiment, we assumed that $\mathtt{N_1}$ was small. Therefore, to maximize parallelism and locality, we stripmined the $\mathtt{I_2}$ loop and then interchanged with the $\mathtt{I_1}$ loop. The result is shown in Loop Nest 9. For both ORIG and PLAS versions, the Fortran-S compiler then distributed iterations of the innermost loop across processors.

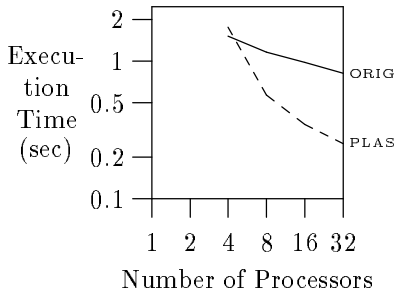


(a) Problem Size: N_2=5000

(b) Problem Size: N_2=10000



(c) Problem Size: N_2=15000



(d) Problem Size: N_2=20000

Figure 4: *Execution times for original and optimized versions of* DMXPY *(Loop Nest 8).* $N_1 = 10$.

Loop Nest 8

```
      /* DMXPY (original) */

      DO I1 = 0 TO N1
         DO I2 = 0 TO N2
R:          Y[I2] = Y[I2] + X[I1] * M[I2,I1]
         END DO
      END DO
```

Loop Nest 9

```
      /* DMXPY (optimized for locality and parallelism) */

      niter = ceil(N2/P)
      DO I2' = 0 TO P
         DO I1 = 0 TO N1
            DO I2 = niter*I2' TO MIN(niter*(I2'+1)-1, N2)
R:             Y[I2] = Y[I2] + X[I1] * M[I2,I1]
            END DO
         END DO
      END DO
```

Figure 4 depicts the performance of the ORIG and PLAS versions of these programs for four different problem sizes. As can be seen in Figure 4(a), the overhead for applying PLAS was less than 10% of the sequential execution time. In this version, no attempt was made to reduce loop overhead. We expect that optimizations could reduce this overhead significantly. Nonetheless, in general, the optimized version significantly outperformed ORIG. The only exception occurred when the number of processors was very small, in which case the degree of false sharing was too small to offset the load imbalance caused by PLAS. This effect can be seen in Figure 4(c)(d). Note that the curves corresponding to the optimized version are

smoother as well. This makes the performance of the optimized versions easier to predict, which facilitates program tuning.

6.2 TRIANGLE

In DMXPY, the write reference pattern is the same during each each iteration of the I_1 loop. is the same on every execution of this loop. Thus, the degree of false sharing is not very high. False sharing would have been more significant if the reference pattern were changed across executions of the I_1 loop. To test the benefits of our techniques under higher degrees of false sharing we created an artificial benchmark TRIANGLE (Loop Nest 10) with exactly this property. The outer loop was executed serially. Iterations of the inner loop were distributed across processors.

Loop Nest 10

```
        /* TRIANGLE */

        DO I1 = 0 TO N1
           DO I2 = I1+1 TO N2
R:            Y[I2] = Y[I2] + X[I1] * M[I2,I1]
           END DO
        END DO
```

The performance of TRIANGLE can be seen in Figure 5. Again, the PLAS version outperformed the ORIG version. Because the lower bound of the inner loop depended on the outer loop index, the ORIG version lost affinity across iterations of the outer loop. version lost reuse across executions of the inner loop. One positive side effect of the PLAS optimization is that it created reuse opportunities across executions of the inner loop.

6.3 LLK18

Figure 6 presents execution times for unoptimized and optimized versions of LLK18, a two-dimensional explicit hydrodynamics code, known as Lawrence Livermore Kernel 18. This code contained three loop nests similar to that depicted in Loop Nest 4. Although each loop nest contained multiple write references, but the references within each loop nest belonged to the same reference group. This is because the corresponding arrays had the same dimensions and subscript expressions, and the Fortran-S compiler automatically page-aligned arrays, when possible. Therefore, if false sharing was eliminated with respect to one write reference in each loop nest, it was automatically eliminated with respect to both. To maximize parallelism and locality, we applied the same optimizations as we did to DMXPY. The Fortran-S compiler then distributed iterations of the innermost loop across processors.

The performance of the ORIG and PLAS versions can be seen in Figure 6. Note that, as the number of processors was increased past a threshold, the performance of PLAS more or less flattened out. This is true

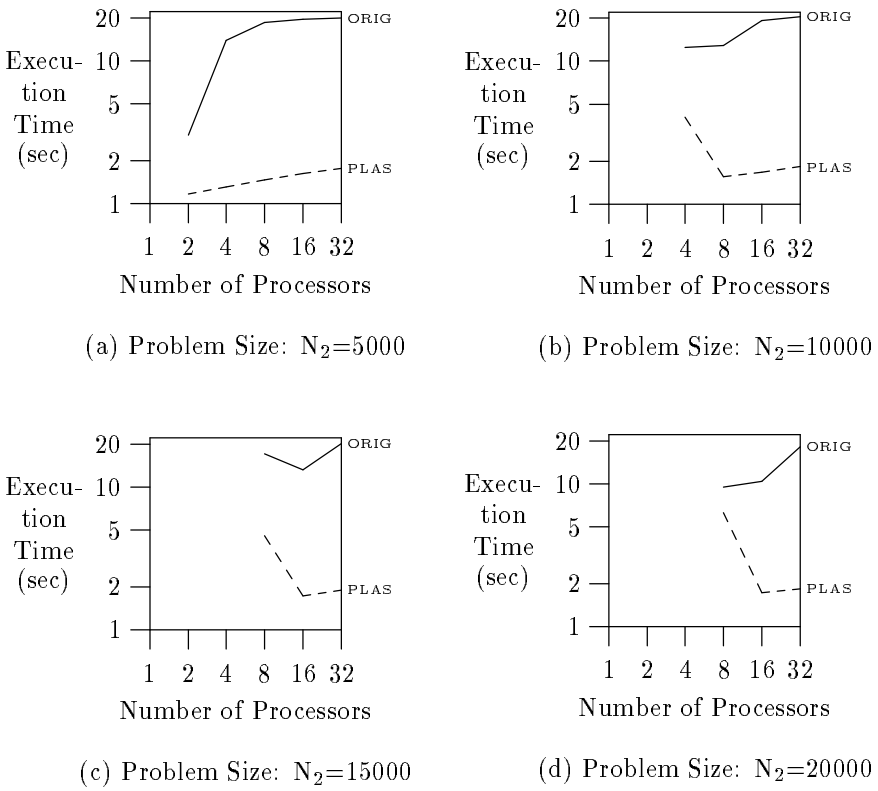


Figure 5: *Execution times for unoptimized and optimized versions of* TRIANGLE *(Loop Nest 10).* $\mathtt{N_1} = 100$.

for all four graphs in Figure 5 as well. This trend is due largely to the constraints that PLAS imposed on the scheduling policy. In general, with any program, increasing parallelism past some threshold will cause performance to worsen. Finding this point, however, is non-trivial. Because PLAS required that pages were treated as indivisible units (i.e., all writes to a given page had to be performed by the same processor), the maximum amount of parallelism was bounded from above by the number of pages. Therefore, PLAS has the side effect of bounding the amount of parallelism that could be exploited.

The best example of this effect can be seen in Figure 6(a), where performance more or less flattened out after 8 processors, increasing only slightly beyond this point. The flattening out occurred because at most 8 processors were used, even if more were available. The slight but steady increase after this point occurred for two reasons. First, in the current version of the compiler, no attempt was made to prevent the execution of empty loop iterations. Second, the program was forked across all available processors, regardless of whether they were used. Both of these could have been overcome at least partially in a more mature compiler, in which case performance would have been expected to level out even more. Had we been able to run experiments on larger systems, we would have expected to see this same trend in the other graphs in Figure 6 as well.

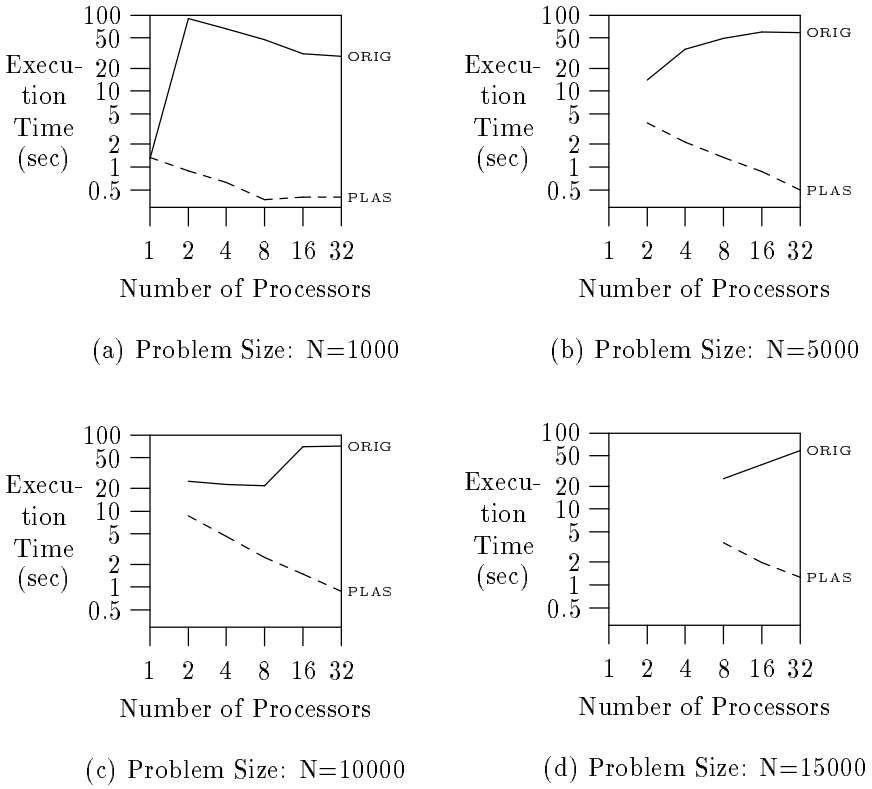


Figure 6: *Execution times for original and optimized versions of* LLK18. *All arrays were page-aligned so there was only one reference group per loop nest.*

6.4 Key Loop Nest from LLK18 with Multiple Reference Groups

To test our extensions for handling loop nests containing multiple reference groups, we changed the bounds of array `ZV` from Loop Nest 4 so that they no longer match those of `ZU`. To maximize parallelism and locality, we again applied the same optimizations as we did to DMXPY. Again, the Fortran-S compiler then distributed iterations of the innermost loop across processors. The results are shown in Figure 7.

In general, for 2 to 32 processors, the optimized version greatly outperformed ORIG. One advantageous side effect of PLAS is that page-level locality was increased and the working set size was decreased. Occasionally the effects are dramatic. For example, in Figure 7(c), ORIG performed very poorly on two processors because of thrashing. Because of the smaller working set of the optimized version, it performed much better.

7 Related Research

The potential performance degradation that can be caused by false sharing of array data has been studied by several researchers. Based on this research, data layout optimizations padding [BFS89, LP92, TLH92, AL93, AALT94, Mon95] have been proposed. Others have studied data layout optimizations to reduce false

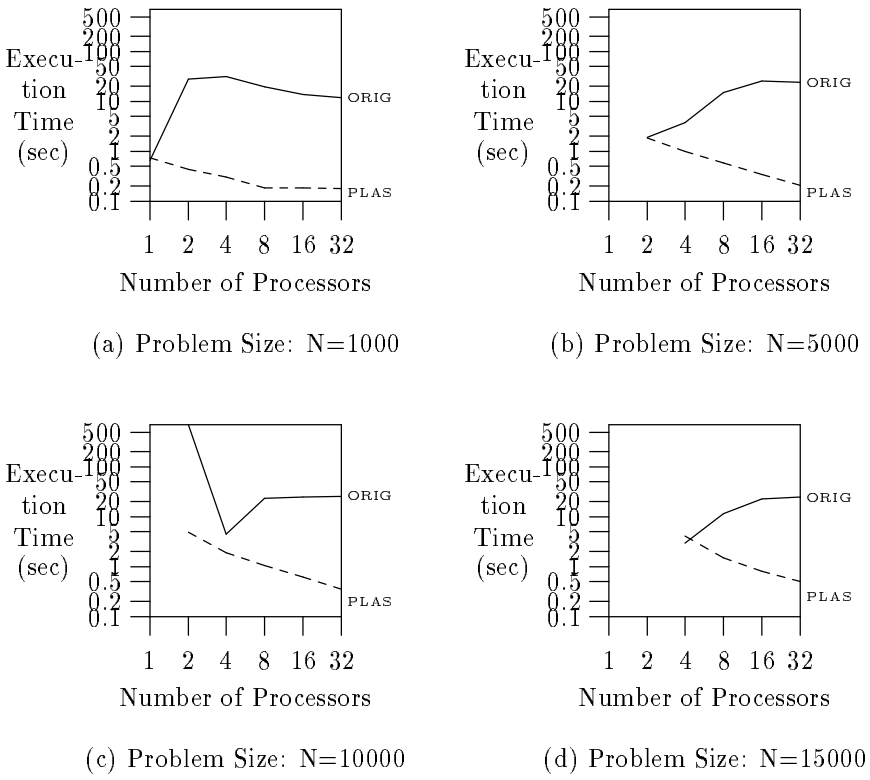


Figure 7: *Execution times for original and optimized versions of Loop Nest 4 from* LLK18.`ZU` *and* `ZV` *were declared with different dimensions, so that Loop Nest 4 would contain references from two distinct reference groups.*

sharing in languages with structures and pointers [EJ91].

In many cases, when coherency units are small, compiler-directed program transformations that increase temporal and spatial locality without directly considering the size of the coherency unit alleviate much of the problem. These include transformations such as loop interchanging that increase locality within an individual loop nest [BEJW92, KM92, WL91] as well as transformations that increase locality across loops, for example [HA90, AHD93]. Unfortunately, these transformations achieve affinity at the data-level only. When the coherency unit becomes larger, such techniques no longer suffice.

An alternate compile-time approach that we explored attacks ping-pong effects only [BGM94]. This approach alleviates ping-pong effects by batching up write requests which encourages processors to perform multiple writes to a page before relinquishing the page. This transformation is simpler to implement and can be applied in more cases than the transformation described here, but yields a smaller performance improvement and only when the amount of parallelism is moderate.

Other researchers, for example [CGL+93, ACIK93, KNS94, AFMP95], have looked at using a block-cyclic owner computes rule to compile data-parallel languages such as HPF [KLS+94]. Some of these techniques are also based on generating and solving sets of inequalities. However, none of them have considered the

approach of precomputing solutions to reduce compile-time overhead.

Run-time solutions for eliminating ping-pong effects have also been proposed. One approach is to relax the consistency model. For example, systems such as Treadmarks [KDCZ94] (by default) and KOAN [LP92] (as an option) allow multiple copies of writable pages to exist and merge modifications only at synchronization points. While these run-time techniques are more general than the compile-time techniques that we study here, they entail a significant space cost to keep track of modifications as well as a time cost associated with both the bookkeeping and the merging. Most of this cost disappears if only single writer false sharing is present. Because we believe that this is a nice solution for single writer false sharing, we have limited our focus to multiple writer false sharing.

For this study, we targeted the elimination of false sharing to improve performance. Because of our assumptions of a page-coherent system (supported in either hardware or by the run-time system), the resulting program would execute correctly regardless of whether false sharing was eliminated. In contrast, on systems where no hardware or run-time support for coherence is provided, false sharing *must* be eliminated to ensure correctness. Breternitz et al. [BLSS93] study this problem. Consequently, their techniques are more general but likely to be less efficient.

8 Conclusions

In this paper, we have presented a transformation for eliminating false sharing within loop nests by applying a page-based owner-computes rule. By using the same page-to-processor mapping across loops, we can achieve page-level affinity across loop nests as well. The constraints that must be met can be expressed as a set of inequalities and solved via standard solution techniques. Computation can be done symbolically at compile time if necessary. Consequently, we can precompute the solution in many commonly occurring cases to reduce compile-time overhead.

Although our goal was to eliminate false sharing, there are two beneficial side effects to our technique. First, locality is improved both within and across loops. Second, in some cases, our technique can reduce the number of barriers that are needed, thereby further improving performance.

Although the derivation of loop bounds is more complex than in conventional blocking, our experimental results have shown that run-time overhead is generally low and quickly offset as the number of processors is increased to even a moderate number. An additional benefit to our technique is that performance generally becomes more predictable, which facilitates both manual and automatic program tuning.

Because the performance degradation due to page migrations is proportional to page size, so is the benefit of applying our techniques. Consequently, we expect the performance results obtained under the KOAN DSM system to be realizable under other DSM systems with comparable page sizes. Although we target systems with page-sized coherency units it might also be possible to realize smaller performance gains

on DSM systems such as the Kendall Square Research KSR1 and KSR2 which support smaller coherency units.

References

[AALT94] Saman P. Ammarsinghe, Jennifer M. Anderson, Monica S. Lam, and Chau-Wen Tseng. Design and Evaluation of Compiler Optimizations for Scalable Address Space Machines, 1994. To be published.

[ACIK93] Corinne Ancourt, Fabien Coelho, François Irigoin, and Ronan Keryell. A Linear Algebra Framework for Static HPF Code Distribution. In *Proceedings of the Fourth Workshop on Compilers for Parallel Computers*, Delft, The Netherlands, December 1993.

[AFMP95] A. André, M. Le Fur, Y. Mahéo, and J.-L. Pazat. The Pandore Data Parallel Compiler and its Portable Runtime. In *HPCN Europe'95*, Milan, Italy, May 1995. To appear in LNCS, Springer Verlag.

[AHD93] Bill Appelbe, Charles Hardnett, and Sri Doddapaneni. Program Transformation for Locality Using Affinity Regions. In *the* Sixth Annual Workshop on Languages and Compilers for Parallel Computing, Portland, Oregon, August 1993. Published in *Languages and Compilers for Parallel Computing*, Banerjee et al. (Eds.), LNCS 768, Springer–Verlag, 1994, pages 290–300.

[AI91] C. Ancourt and F. Irigoin. Scanning Polyhedra with Do Loops. In *Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 39–50, Williamsburg, Virginia, April 1991.

[AL93] Jennifer Anderson and Monica Lam. Global Optimizations for Parallelism and Locality on Scalable Parallel Machines. In *Proceedings of the SIGPLAN '93 Conference on Programming Languages Design and Implementation*. ACM Press, June 1993.

[BCZ92] S. Benkner, B. Chapman, and H. Zima. Vienna Fortran 90. In *Scalable High Performance Computing Conference*, pages 51–59. IEEE Computer Society Press, April 1992.

[BEJW92] François Bodin, Christine Eisenbeis, William Jalby, and Daniel Windheiser. A Quantitative Algorithm for Data Locality Optimization. In *Code Generation-Concepts, Tools, Techniques*. Springer-Verlag, 1992.

[BFS89] William J. Bolosky, Robert P. Fitzgerald, and Michael L. Scott. Simple But Effective Techniques for NUMA Memory Management. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 19–31. ACM Press, December 1989.

[BGM94] François Bodin, Elana D. Granston, and Thierry Montaut. Evaluating Two Loop Transformations for Reducing Multiple-Writer False Sharing. In the *Seventh Annual Workshop on Languages and Compilers for Parallel Computing*, Ithaca, New York, August 1994. Published as LNCS 892, pages 423–439, Pingali et al. (Eds.), 1995. Springer-Verlag, Berlin, Heidelberg.

[BKP93] F. Bodin, L. Kervella, and T. Priol. Fortran-S: A Fortran Interface for Shared Virtual Memory Architectures. In *Supercomputing '93*, pages 274–283. IEEE Computer Society Press, November 1993.

[BLSS93] Mauricio Breternitz, Jr., Michael Lai, Vivek Sarkar, and Barbara Simons. Compiler Solutions for the Stale-Data and False-Sharing Problems. Technical Report 03.466, IBM Santa Teresa Laboratory, April 1993.

[CF78] L.M. Censier and P. Feautrier. A New Solution to Coherence Problems in Multicache Systems. *IEEE Transactions on Computers*, pages 1112–1118, December 1978.

[CGL+93] Siddhartha Chatterjee, John R. Gilbert, Fred J. E. Long, Robert Schreiber, and Shun-Hua Teng. Generating Local Address Communication Sets for Data-Parallel Programs. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice Of Parallel Programming*, pages 149–158, San Diego, California, 1993.

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

[EJ91] Susan J. Eggers and Tor E. Jeremiassen. Eliminating false sharing. In *Proceedings of the International Conference on Parallel Processing*, pages 377–381. CRC Press, Inc., August 1991.

[Fea89] P. Feautrier. Semantical Analysis and Mathematical Programming, Application to Parallelization and Vectorization. In M. Cosnard et al., editor, *Parallel and Distributed Algorithms*, pages 309–320. Elsevier Science Publishers, 1989.

[FSHK+91] G. Fox, K. Kennedy S. Hiranandi, C. Koebel, U. Kremer, C. Tseng, and M. Wu. Fortran D Language Specification. Technical Report TR-90079, Department of Computer Science, Rice University, March 1991.

[Fur95] Marc Le Fur. Scanning Parameterized Polyhedron using Fourier-Motzkin Elimination. In *HPCS'95*, Montréal, Canada, July 1995. (to appear).

[Gra93] Elana D. Granston. Toward a Compile-Time Methodology for Reducing False Sharing and Communication Traffic in Shared Virtual Memory Systems. In the Proceedings of the *Sixth Annual Workshop on Languages and Compilers for Parallel Computing*, Portland, Oregon, August 1993. Published as LNCS 768, pages 273–289, Banerjee et al. (Eds.), 1994. Springer-Verlag, Berlin, Heidelberg.

[HA90] David E. Hudak and Santosh G. Abraham. Compiler Techniques for Data Partitioning of Sequentially Iterated Loops. In *Proceedings of the International Conference on Supercomputing*, pages 187–200. ACM Press, June 1990.

[JC93] T. Risset J.F. Collard, P. Feautrier. Construction of DO Loops from Systems of Affine Constraints. In *LIP Research Report 93-15*, May 1993.

[KDCZ94] P. Keleher, S. Dwarkadas, A. Cox, and W. Zwaenepoel. Treadmarks: Distributed Shared Memory On Standard Workstations and and Operating Systems. In *Winter Usenix Conference*, 1994.

[KLS+94] Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele Jr., and Mary E. Zosel. *The High Performance Fortran handbook.* MIT Press, Cambridge, Massachusetts, 1994.

[KM92] Ken Kennedy and Katheryn S. McKinley. Optimizing for Parallelism and Data Locality. In *International Conference on Supercomputing*, pages 323–334. ACM Press, July 1992.

[KNS94] Ken Kennedy, Nenad Nedeljkovic, and Ajay Sethi. Efficient Address Generation for Block-Cyclic Distributions. Technical report, Center for Research on Parallel Computation, Rice University, Technical Report No. CRPC-TR94487-S, Houston, Texas, December 1994.

[Li86] Kai Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors.* PhD thesis, Yale University, September 1986.

[LP92] Z. Lahjoumri and T. Priol. KOAN: A Shared-Memory for the iPSC/2 Hypercube. In *CONPAR/VAPP92*, LNCS 634. Springer-Verlag, September 1992.

[Mon95] Thierry Montaut. *Méthodes pour l'élimination du faux-partage et l'optimisation de la localité pour mémoire virtuelle partagée.* PhD thesis, IRISA, Campus de Beaulieu, 1995. In preparation.

[OKB] M.F.P. O'Boyle, L. Kervella, and F. Bodin. Synchronization Minimization in a SPMD Execution Model. To appear in the *Journal of Parallel and Distributed Computing.*

[Pug91] William Pugh. The Omega Test: A Fast and Practical Integer Programming Algorithm for Dependence Analysis. In *Supercomputing*, 1991.

[Sch86] Alexander Schrijver. *Theory of Linear and Integer Programming.* Wiley-Interscience, 1986.

[TLH92] Josep Torrellas, Monica S. Lam, and John L. Hennessy. False Sharing and Spatial Locality in Multiprocessor Caches, August 1992. Submitted to *IEEE Transactions on Computers.*

[WL91] Michael E. Wolf and Monica S. Lam. A Data Locality Optimizing Algorithm. In *Proceedings of the SIGPLAN '91 Conference on Programming Languages Design and Implementation*, pages 30–44. ACM Press, June 1991.