Loop Transformations to Prevent False Sharing

Francois Bodin
Elana Granston
Thierry Montaut

CRPC-TR95528
May 1995

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

Loop Transformations to Prevent False Sharing

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

François Bodin†

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

May 16, 1995

To appear in the *International Journal of Parallel Programming*

Abstract

To date, page management in shared virtual memory (SVM) 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, a loop transformation theory is developed for identifying and eliminating potential sources of multiple-writer false sharing and other sources of page migration resulting from regular references in numerical applications. Loop nests of one and two dimensions (before blocking) with single-level, DOALL-style parallelism are covered. The potential of these transformations is demonstrated experimentally.

Key words: false sharing, page-level affinity scheduling, loop transformations, shared virtual memory.

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 is not easy. Writing correct compilers, let alone good compilers, is not easy either. Typically, compiler technology lags years behind architectural innovations.

To simplify the programming of and compiling for such systems, much research effort has been directed toward the implementation of shared virtual memory (SVM) interfaces [Li86] which provide the programmer with the illusion of a global address space. In such a system, the unit of data to which coherency is applied

*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.

†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.

Loop Nest 1

```
DO I1 = 0 TO N1-1
  DOALL I2 = 0 TO N2-1
    A[2 * I1 + 10 * I2 + 1] = h(I1, I2)
  END DOALL
END DO
```

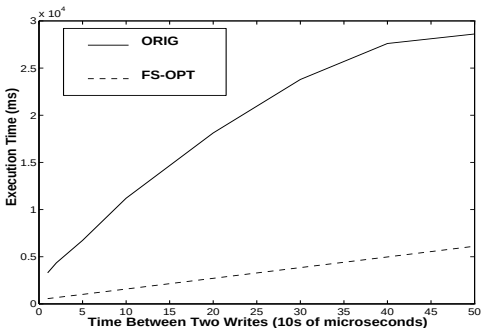


Figure 1: *Execution times for two versions of Loop Nest 1.*

is termed a *page*. In practice, this coherency unit may be an actual physical page, a cache line, or a multiple or portion thereof.

The success of the SVM 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 of pages, particularly multiple-writer false sharing, which arises when two or more processors are writing distinct data on the same page in an unsynchronized fashion. Assume that the system supports an invalidation-based coherence protocol whereby, before a processor can write to a page, all other copies must be invalidated. Then multiple-writer false sharing causes the writes to be serialized. In the best case, the number of page migrations that this causes is one less than the number of processors sharing the page. Often, however, this number rises much higher due to the repetitive bouncing of this page between processors. This latter phenomenon is known as *ping-pong effects*. This can result in unacceptably high communication costs, traffic levels and memory access delays.

Intuitively, false sharing effects increase as the number of processors increases. Less intuitively, false sharing effects can also increase as the time between successive writes by a single processor increases. For example, in an invalidation-based protocol, the longer the time between successive writes, the greater the likelihood that a page will be invalidated in between a successive pair of writes to that page, thereby increasing ping-pong effects and preventing the exploitation of page-level locality. This effect can be seen in Figure 1 which presents execution times for two versions of Loop Nest 1 as a function of the time between successive writes by a given processor:

- the original loop (ORIG),
- a version optimized to prevent multiple-writer false sharing (FS-OPT).

Both versions were executed on 16 processors of a 32-processor iPSC/2 under the KOAN SVM system [LP92],

which supports the aforementioned invalidation-based coherence protocol and employs a page size of 4 KB (512 double-precision numbers). The problem size was $\mathtt{N_1} = \mathtt{N_2} = 10^3$. The performance difference between the ORIG and FS-OPT versions demonstrates the performance degradation that can be caused by false sharing. The effect of varying the number of processors can be seen in the performance results presented in Section 8.

In this paper, a loop transformation theory is presented for identifying and eliminating potential sources of multiple-writer false sharing at compile time. This theory provides the basis for the transformation used in the above experiment. Loop nests of one and two dimensions (before blocking) with single-level, DOALL style parallelism are discussed. We also target the identification and elimination of potential sources of page-level sharing that arise from writes by multiple processors across DOALL loops.

The remainder of this paper is organized as follows. Section 2 provides an overview of page-level sharing. Section 3 discusses the basic theory in terms of single loop containing a single static write reference to a one-dimensional array that is a function of a single loop index. Sections 4 and 5 extend the theory to two-dimensional loop nests. Section 6 extends the theory to multi-dimensional arrays. Section 7 briefly addresses extensions for handling loops containing multiple static write references. Section 8 presents experimental results. Section 9 discusses related research. Section 10 concludes this paper.

2 Page-level Sharing

Consider the following simple DOALL loop before (left) and after (right) blocking:

Loop Nest 2

```
DOALL I = 0 TO N-1
      A[3*I] = h(I)
END DOALL
```

Loop Nest 3

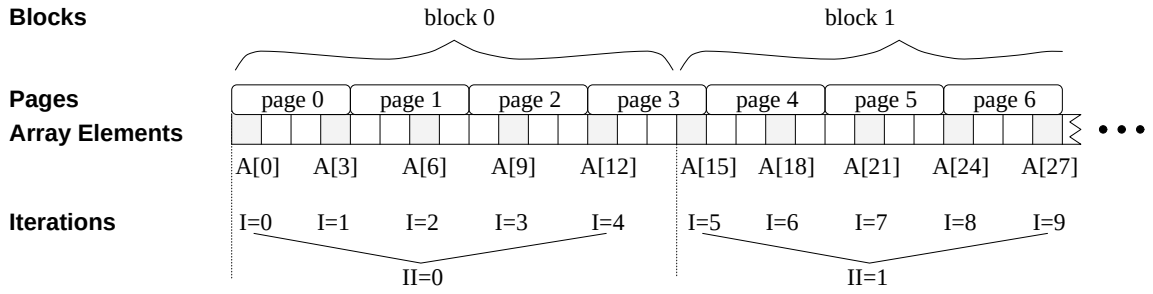
```
DOALL II = 0 TO ⌈N/β⌉ − 1
   DO I = II*β TO MIN((II+1)*β-1, N-1)
      A[3*I] = h(I)
   ENDDO
END DOALL
```

Assume that a page can hold precisely m array elements.

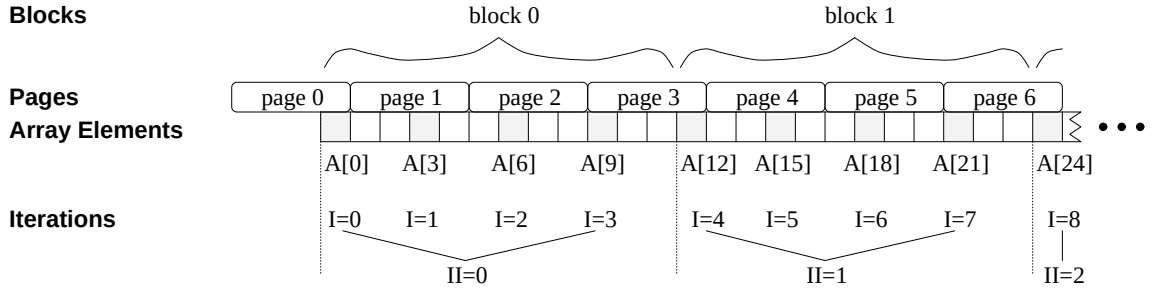
Example 1 If the *block size* β in Loop Nest 3 is not a multiple of $m/3$, multiple processors will be writing simultaneously to different elements of A on the same page, thus resulting in false sharing. Depicted below is the mapping between pages and DOALL loop iterations that results when $m = 4$ and $\beta = 5$. Note that $5 \neq k\,(4/3)$ for any $k \in \mathbb{P}$ [1]. Because the elements on page 3 are accessed during distinct DOALL loop iterations, page 3 may be falsely shared[2].

[1] In this paper, we use the following set notation: $\mathbb{Z}$ is the set of integers, $\mathbb{P}$ is the set of positive integers, $\mathbb{N}$ is the set of non-negative integers, and $\mathbb{Q}$ is the set of rational numbers.

[2] For this and similar examples, a small page size was chosen to simplify illustrations. Were the page size larger, there would be more such cases.



Example 2 For this same loop, false sharing also results when β is a multiple of $m/3$, if `A[0]` is not aligned with the beginning of a page. Depicted below is the mapping between pages and DOALL loop iterations that results when $m = 4, \beta = 3\,(4/3) = 4$, but `A[0]` is positioned at the end of a page. Depending on the assignment of blocks to processors, pages 3 and 6 may be falsely shared.



When false sharing occurs, no two processors are accessing the same datum, so no classical, loop-carried dependence exists. Therefore, such sharing generally does not get considered in traditional dependence-analysis based optimization frameworks.

In Examples 1 and 2 above, the need for page-migrations or remote accesses arose from multiple references to the same page within a single execution of a DOALL loop. Suppose instead that a DOALL loop were enclosed within a serial loop. Then such a need could also arise if there were overlap between the sets of pages accessed during distinct executions of the DOALL loop (equivalently, distinct iterations of the enclosing loop). In this paper, we target the prevention of this source of page-level sharing as well. We can also extend our techniques to prevent migrations across distinct DOALL loops.

3 Detecting and Preventing False Sharing in One-Dimensional Loop Iteration Spaces

Consider the following loop nest containing a reference R to $\mathtt{A}[\mathtt{c}\,\mathtt{I} + \lambda]$, where `I` is the index of the enclosing loop, and $\mathtt{c}, \lambda \in \mathbb{Z}$ are invariant with respect to the loop.

Loop Nest 4

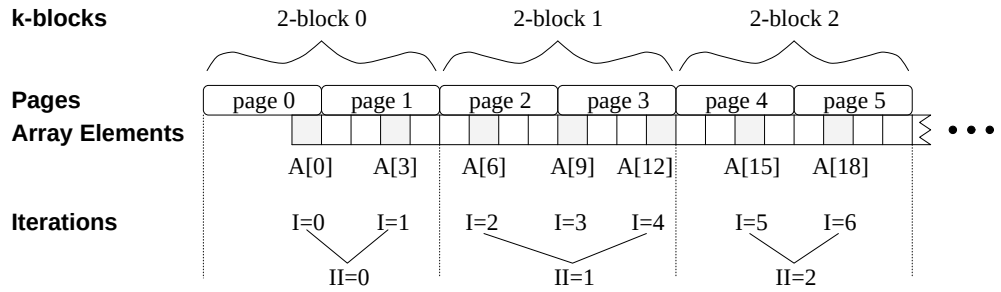


Figure 2: Example of preventing false sharing within Loop Nest 5 when R is a reference to `A[3*I]`. $o($`A[0]`$) = 3$, $m = 4$, $k = 2$, $n = 0$, $\beta = \beta(2) = 8/3$, and $\phi = \phi(0) = 0$.

```
     DOALL I = 0 to N-1
R:         A[c*I+λ] = h(I)
     END DOALL
```

In this loop nest, as in all other loop nests in this paper, it is assumed that standard optimizations to remove loop-carried dependences (e.g., as scalar expansion) and those to improve locality have already been applied where possible and profitable.

Intuitively, multiple-writer false sharing can be prevented by partitioning pages into blocks of k pages each, known as *k-blocks*. Then we can partition the computation into chunks. Each chunk includes the `I`-loop iterations that map to a given k-block. (An iteration *maps* to a *k-block* if the element of `A` written during that iteration lies within that *k-block*.) Chunks are then assigned to processors indivisibly. The result of this transformation is depicted as Loop Nest 5[3]. With appropriate constraints on β and ϕ, false sharing is prevented.

Loop Nest 5

```
     DOALL II = 0 TO ⌈(N + φ)/β⌉-1
        DO I = MAX(⌈II * β − φ⌉, 0) TO MIN(⌈(II + 1) * β − φ⌉-1, N-1)
R:         A[c*I+λ] = h(I)
        END DO
     END DOALL
```

Example 3 Assume that Loop Nest 5 contains the reference R to `A[3*I]`. Suppose that a page contains precisely m=4 elements of `A`, and that `A` is laid out across pages so that element `A[0]` is the last element on some page. Choosing $\beta = 8/3$ and $\phi = 0$ partitions pages into 2-page blocks and then partitions iterations accordingly, as shown in Figure 2.

For example, `A[9]` and `A[12]`, which lie on page 3 are accessed during iterations `I`=3 and `I`=4, respectively. Page 3 is mapped to *k-block* 1. All iterations that map to *k-block* 1 are executed during iteration

[3] In the remainder of this paper, it will assumed that β and ϕ may be rational and that $\phi < \beta$. The reason for this will become apparent later in section.

II=1 of the DOALL loop. Thus, both of these write accesses will be performed by the processor, thereby preventing false sharing of page 3.

In the remainder of this section, we present our false sharing prevention technique more rigorously. While the presentation may seem unnecessarily formal for the relatively straightforward case of one-dimensional loop nests, the results presented in this section form a foundation that is necessary for extending these ideas to two-dimensional loop nests (Section 4).

3.1 Preliminaries

Let $p(\mathtt{A[expr]})$ be the number of the page containing $\mathtt{A[}expr\mathtt{]}$. Arbitrarily let $p(\mathtt{A[0]}) = 0$. Let $o(\mathtt{A[expr]})$ be the offset of $\mathtt{A[}expr\mathtt{]}$ on a page, where $0 \leq o(\mathtt{A[expr]}) < m$. (Recall that m is the number of array elements that reside on a page.) $o(\mathtt{A[0]})$ can be determined directly from the starting address of A.

Lemma 1 *Based on the above assumptions,*

(a) $o(\mathtt{A[expr]}) = (o(\mathtt{A[0]}) + expr) \bmod m.$

(b) $p(\mathtt{A[expr]}) = \lfloor \frac{o(\mathtt{A[0]}) + expr}{m} \rfloor.$

Let $sign(\mathtt{c})$ be 1 if $\mathtt{c} > 0$, -1 if $\mathtt{c} < 0$, and 0 otherwise. Let $sign^{-}(\mathtt{c}) = max(-sign(\mathtt{c}), 0)$.

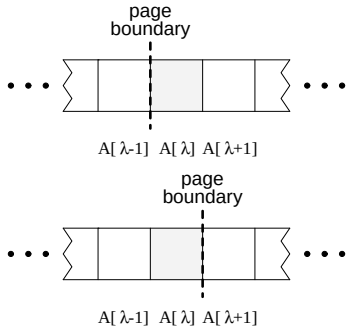
Definition 1 *Let R be a reference to $\mathtt{A[c * I + \lambda]}$, where $\mathtt{c}, \lambda \in \mathbb{Z}$ and I is the index of a normalized enclosing loop.* A *is* **perfectly aligned** *with respect to R if and only if*

$$o(\mathtt{A}[\lambda]) \;=\; (-sign^{-}(\mathtt{c})) \bmod m \;=\; \begin{cases} 0 & \text{if } \mathtt{c} \geq 0 \\ m-1 & \text{otherwise.} \end{cases}$$

Intuitively, given Loop Nest 4, A is perfectly aligned with respect to R if and only if

- $\mathtt{c} \geq 0$ and array A is laid out across pages so that the element of A accessed in iteration $\mathtt{I} = 0$ is located at the *beginning* of a page, *or*

- $\mathtt{c} < 0$ and array A is laid out across pages so that the element of A accessed in iteration $\mathtt{I} = 0$ is located at the *end* of a page.



3.2 Computing Blocking Factors

Suppose that Loop Nest 4 has been stripmined to exploit spatial locality:

Loop Nest 6

```
      DOALL II = 0 TO ⌈N/β⌉ − 1
         DO I = ⌈II ∗ β⌉ TO min(⌈(II + 1) ∗ β⌉-1, N-1)
R:          A[c*I+λ] = ···
         END DO
      END DOALL
```

Which values of β will prevent false sharing?

Let $i \in \mathtt{I}$ denote a value that variable I can take on during execution of the loop nest. Let $i \in \mathtt{I}(\mathtt{II} = ii)$ denote a value that variable I can take on when index II of the enclosing loop has value ii.

Definition 2 *An iteration $i \in \mathtt{I}$* **maps** *to a page p with respect to a reference R, if the element of* A *referenced at R during iteration i is located on page p.*

Lemma 2 *When $\mathtt{c} \neq 0$, the iterations of the* I*-loop that map to a given page fall within an interval of size $m/|\mathtt{c}|$.*

Consider the reference R to $\mathtt{A}[\mathtt{c} * \mathtt{I} + \lambda]$ in Loop Nest 6. Note that c must be non-zero. Otherwise, parallelization of this loop would have been illegal.

Lemma 3 *If $0 < |\mathtt{c}| < m$, then false sharing might exist. If $|\mathtt{c}| \geq m$, then there is no page-level sharing of any kind between processors.*

The remainder of this section focuses on preventing the false sharing that arises when $0 < |\mathtt{c}| < m$. However, the theory presented here is sufficiently robust to encompass the case where $|\mathtt{c}| \geq m$ as well, as this will become useful when analyzing two-dimensional loop nests (Section 4).

Theorem 1 *Assuming that* A *is perfectly aligned with respect to R, employing a block size of*

$$\beta \in \text{BLK\_SZ}(\mathtt{c}, m) = \begin{cases} \{\ \beta(k) = k\,\beta^{min} \mid k \in \mathbb{P},\ \beta^{min} = \frac{m}{|\mathtt{c}|}\ \} & 0 < |\mathtt{c}| < m \\ (0 : \infty]_{\mathbb{Q}} & |\mathtt{c}| \geq m. \end{cases}$$

during the execution of Loop Nest 6 will prevent page-level sharing between processors[4]*.*

Proof Consider the case when $0 < |\mathtt{c}| \leq m$, so $\beta = \beta(k)$. Assume that $ii, ii' \in \mathtt{II}$, $ii < ii'$, and that $i \in \mathtt{I}(\mathtt{II} = ii)$ and $i' \in \mathtt{I}(\mathtt{II} = ii')$. When $\mathtt{c} > 0$, $o(\mathtt{A}[0])+\mathtt{c}*i+\lambda < o(\mathtt{A}[0])+\mathtt{c}*(ii'*\beta)+\lambda \leq o(\mathtt{A}[0])+\mathtt{c}*i'+\lambda$. By the definition of perfect alignment, there exists $t \in \mathbb{Z}$ such that $o(\mathtt{A}[0]) + \lambda = t * m$. Therefore,

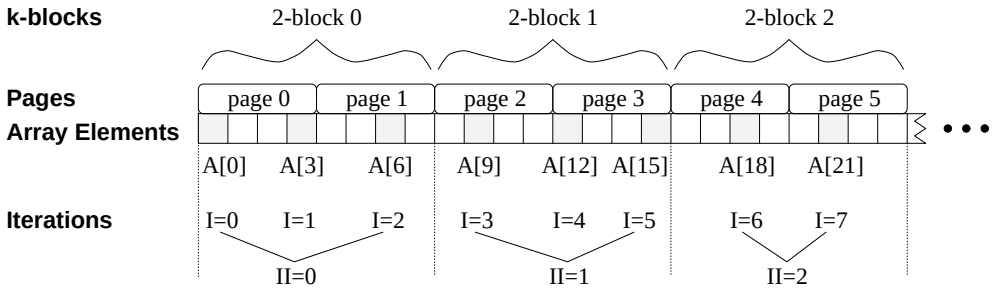
$$p(\mathtt{A}[\mathtt{c} * i + \lambda]) = \left\lfloor \frac{o(\mathtt{A}[0])+\mathtt{c}*i+\lambda}{m} \right\rfloor < ii' * k + t \leq \left\lfloor \frac{o(\mathtt{A}[0])+\mathtt{c}*i'+\lambda}{m} \right\rfloor = p(\mathtt{A}[\mathtt{c} * i' + \lambda]).$$

[4] The notation $(r_{min} : r_{max}]$ denotes the set $\{r \mid r_{min} < r \leq r_{max}, r \in \mathbb{R}\}$. $(r_{mn} : r_{mx}]_S$ denotes the set $(r_{min} : r_{max}] \cap S$, where $S \in \{\mathbb{Z}, \mathbb{P}, \mathbb{N}, \mathbb{Q}\}$.

When $\mathtt{c} < 0$, $o(\mathtt{A}[\mathtt{0}]) + \lambda = t * m - 1$, so $p(\mathtt{A}[\mathtt{c} * i + \lambda]) > p(\mathtt{A}[\mathtt{c} * i' + \lambda])$. Therefore, no page is accessed during more one DOALL loop iteration.

The case where $|\mathtt{c}| \geq m$ follows trivially from Lemma 3. □

Example 4 For example, let $R = \mathtt{A}[\mathtt{3} * \mathtt{I}]$, $m = 4$, and $o(\mathtt{A}[\mathtt{0}]) = 0$. We arbitrarily choose $k = 2$, so that the block size is $\beta = \beta(2) = 8/3$.



Because $k = 2$, the `I`-loop iterations that are executed during any given iteration of the `II`-loop (the outer loop after stripmining the original DOALL loop) are exactly those that map to some block of k pages, or *k-block*. Because the block size is not an integer, the number of iterations that map to a k-block can vary by one from block to block.

This prevents false sharing as follows. Consider elements `A[12]` and `A[15]`, which both lie on page 3 and are accessed during iterations `I`=4 and `I`=5, respectively. Both iterations `I`=4 and `I`=5 are executed during the same DOALL loop iteration. Thus, false sharing of page 3 is prevented.

3.3 Compensating for Imperfect Alignment

Suppose that `A` is not perfectly aligned. Because iteration $\mathtt{I} = 0$ maps to the middle of a block, employing a block size of $\beta = \beta(k)$ is generally insufficient to prevent false sharing. Therefore, when false sharing exists, in addition to blocking by multiple of β^{min}, the interval of iterations that is executed during the first DOALL loop iteration must be shortened to compensate for the initial partial page. We show that this can be achieved by transforming Loop Nest 4 as shown in Loop Nest 5, with the constraints on β and ϕ derived in this section.

Definition 3 *Two loop nests are* **equivalent** *if executing one loop nest yields the same result as executing the other.*

Lemma 4 *Given any $\phi \in [0 : \beta)_{\mathbb{Q}}$, Loop Nest 5 is equivalent to Loop Nest 4 (the original loop).*

Proof Loop Nest 5 is equivalent to Loop Nest 4 if there is a one-to-one correspondence between the values that `I` takes on during execution of Loop Nest 5 and the set of integers $\{0, 1 \ldots, \mathtt{N} - 1\}$. Let $\mathcal{R}(\phi, \beta, ii) =$

$[\ ii * \beta - \phi : (ii+1) * \beta - \phi\ )_{\mathbb{Z}}$. Then the set of I-loop iterations executed during a given DOALL loop iteration $\mathtt{II} = ii$ is

$$\mathcal{R}(\phi, \beta, ii) \cap [0 : N)_{\mathbb{Z}}.$$

The set of I loop iterations executed over all $ii \in \mathtt{II}$ is

$$\begin{aligned} \bigcup_{ii=\lfloor \phi/\beta \rfloor}^{\lceil (\mathtt{N}+\phi)/\beta \rceil - 1} \left(\mathcal{R}(\phi, \beta, ii) \cap [0 : \mathtt{N})_{\mathbb{Z}}\right) &= \left[\ \left\lfloor \frac{\phi}{\beta} \right\rfloor * \beta - \phi\ :\ \left\lceil \frac{\mathtt{N}+\phi}{\beta} \right\rceil * \beta - \phi\ \right)_{\mathbb{Z}} \cap\ [0 : \mathtt{N})_{\mathbb{Z}} \\ &= \{\ 0, 1, \ldots, \mathtt{N}-1\ \}. \end{aligned}$$

Therefore, each I-loop iteration in the range $\{\ 0, 1, \ldots, \mathtt{N}-1\ \}$ is executed at least once. Trivially, no I-loop iteration is executed more than once within any given II-loop iteration. Moreover, given two distinct II-loop iterations ii and ii', $\mathcal{R}(\phi, \beta, ii) \cap \mathcal{R}(\phi, \beta, ii') = \emptyset$. Therefore, no I-loop iteration is executed during more than one II-loop iteration. Thus, a one-to-one correspondence is established. □

Definition 4 *Let $\beta \in$ BLK_SZ$(\mathtt{c}, m)$. $\phi \in [0 : \beta)_{\mathbb{Q}}$ is an* **alignment factor** *for reference R if and only if using that value of ϕ in Loop Nest 5 in combination with block size β suffices to prevent page-level sharing between processors with respect to R.*

Lemma 5 *If $o(\mathtt{A}[-\mathtt{c} * \phi + \lambda]) = (-sign^{-}(\mathtt{c})) \bmod m$, then page-level sharing in Loop Nest 5 is prevented with respect to R, and ϕ is an alignment factor for R.*

Proof Let $ii, ii' \in \mathtt{II}$, $ii < ii'$, $i = \mathtt{I}(\mathtt{II} = ii)$ and $i' = \mathtt{I}(\mathtt{II} = ii')$. Using similar reasoning as in the proof of Theorem 1, it can be shown that if $o(\mathtt{A}[-\mathtt{c} * \phi + \lambda]) = (-sign^{-}(\mathtt{c})) \bmod m$, then $p(\mathtt{A}[\mathtt{c} * i + \lambda]) \neq p(\mathtt{A}[\mathtt{c} * i' + \lambda])$. Therefore, no page is accessed during more than one DOALL loop iteration. □

Theorem 2 *Possible alignment factors for R include*

$$\phi \in \text{ALIGN\_FACTOR}(o(\mathtt{A}[\mathtt{0}]) + \lambda, \mathtt{c}, k, m) = \begin{cases} \{\ \phi(n) \mid n \in [0 : k)_{\mathbb{N}}\ \} & 0 < |\mathtt{c}| < m \\ [0 : \beta)_{\mathbb{Q}} & |\mathtt{c}| \geq m, \end{cases}$$

where $\phi(n) = \dfrac{sign(\mathtt{c}) * (o(\mathtt{A}[\mathtt{0}]) + n * m + \lambda + sign^{-}(\mathtt{c}))) \bmod (k * m)}{|\mathtt{c}|}$.

Proof For any $n \in \mathbb{N}$, $o(\mathtt{A}[-\mathtt{c} * \phi(n) + \lambda]) = (-sign^{-}(\mathtt{c})) \bmod m$. Therefore, by Lemma 5, any $\phi(n) \in$ ALIGN_FACTOR$(m[0]+\lambda, \mathtt{c}, k, m)$ is an alignment factor. The remainder of the proof follows from Lemma 3 and the definition of alignment factor. □

Corollary 1 *If $\phi(n)$ is an alignment factor, then $\forall\ t \in \mathbb{Z}$, so is $\phi(t * k + n)$.*

Proof $\phi(n) = \phi(t * k + n)$. □

Observation 1 *There are k distinct partionings of pages into k-blocks. The choice of n determines which partitioning is effected by $\phi = \phi(n)$.*

Therefore, without loss of generality, we restrict ourselves to values of n such that $0 \leq n < k$.

Let $o_k^n(\texttt{A[expr]})$ be the offset of `A[`*expr*`]` within the k-block containing it. Let $p_k^n(\texttt{A[expr]})$ be the number of that k-block.

Lemma 6 *Suppose $\beta = b(k)$ and $\phi = \phi(n)$.*
(a) $o_k^n(\texttt{A[0]}) = o(\texttt{A[0]}) + n * m$.
(b) $o_k^n(\texttt{A[expr]}) = (o_k^n(\texttt{A[0]}) + expr) \bmod (k * m)$.
(c) $p_k^n(\texttt{A[expr]}) = \lfloor \frac{o_k^n(\texttt{A[0]}) + expr}{k*m} \rfloor$.

Proof (a) Let $ii \in \texttt{II}$. When an alignment factor of $\phi(n)$ is employed, pages are partitioned into k-blocks such that, if $\texttt{c} > 0$ ($\texttt{c} < 0$), $\texttt{A}[\texttt{c}, (ii * \beta - \phi(n)) + \lambda]$ is the first (last) element in a k-block. Therefore, $o_k^n(\texttt{A}[\texttt{c}\,(ii * \beta - \phi(n)) + \lambda]) = (-sign^{-}(\texttt{c})) \bmod (k * m)$, so $o_k^n(\texttt{A[0]}) = o(\texttt{A[0]}) + n * m$.

(b),(c) Substituting $k * m$ for m, the remainder of the proof follows directly from part (a) and Lemma 1. □

Example 5 Assume again that Loop Nest 5 contains the reference R to `A[3*I]`. Suppose that `A` is laid out as shown in Figure 2, so that `A` is not perfectly aligned. We arbitrarily choose $k = 2$ and $n = 0$, so that $\beta = \beta(k) = 8/3$ and $\phi = \phi(n) = 0$. As can be seen in Figure 2, false sharing of page 3 is prevented. Note that there are $k = 2$ possible pairings of pages into 2-blocks. By choosing $n = 0$, page 0 has become the first page in some 2-block. Had we chosen $n = 1$, page 0 would have been the second page in some 2-block.

As will be seen in Section 8, using non-integer block sizes and alignment factors generally does not add significant overhead. Moreover, much of the time, block sizes will actually end up being integers. If not, however, block sizes can be restricted to integers although this might increase the minimum block size that meets this requirement. Furthermore, whenever the block size is an integer, an integer alignment factor can be found. See Appendix A for more detail.

4 Preventing False Sharing In Two-Dimensional Loop Iteration Spaces

Assume that array subscripts have the form $\texttt{c}_1 * \texttt{I}_1 + \texttt{c}_2 * \texttt{I}_2 + \lambda$, where $\texttt{I}_1$ and $\texttt{I}_2$ are indices of the outer and inner loops, respectively, and $\texttt{c}_1, \texttt{c}_2, \lambda \in \mathbb{Z}$ are invariant with respect to both loops.

Loop Nest 7

```
DO I1 = 0 TO N1-1
   DO I2 = 0 TO N2-1
      A[c1 * I1 + c2 * I2 + λ] = h(I1, I2)
   END DO
END DO
```

In this section, we analyze the case where the inner loop is parallelized as shown below. In the next section, we analyze the case where the outer loop is parallelized.

Loop Nest 8

```
DO I1 = 0 TO N1-1
   DOALL II2 = 0 TO ⌈N2/b⌉ − 1
      DO I2 = ⌈II2 * b⌉ TO MIN(⌈(II2 + 1) * b⌉-1, N2 − 1)
         A[c1 * I1 + c2 * I2+λ] = h(I1, I2)
      END DO
   END DOALL
END DO
```

Preventing page-level sharing between processors is accomplished in two steps. First, we apply the techniques from the previous section to prevent false sharing. This is equivalent to preventing the false sharing that results during a single execution of the DOALL loop (i.e., a single iteration of the $\mathtt{I_1}$-loop). Then, we build on this to prevent sharing across executions of the DOALL loop (i.e., across iterations of the $\mathtt{I_1}$-loop).

4.1 Step 1: Preventing Page-level Sharing within a Single Doall Loop Execution

Because the expression $\mathtt{c_1} * \mathtt{I_1} + \lambda$ is constant within an iteration of the $\mathtt{I_1}$-loop, the distinctions between this case and that presented in the previous section are that (1) the constant expression with respect to the inner loop is now $\mathtt{c_1} * \mathtt{I_1} + \lambda$, rather than λ, and (2) the set of possible alignment factors depends on the value of $\mathtt{I_1}$.

Observation 2 *Within a single DOALL loop execution, (equivalently, a single iteration of the* $\mathtt{I_1}$*-loop) page-level sharing between processors may exist only when* $0 \leq |\mathtt{c_2}| < m$.

Note that $\mathtt{c_2}$ must be non-zero or else parallelization of the inner loop would have been illegal. As in the previous section, we again address only those cases where the coefficient of the inner loop index is non-zero.

Theorem 3 *Within a single DOALL loop execution, we can prevent page-level sharing by employing a blocking factor* $\beta \in \text{BLK\_SZ}(\mathtt{c_2}, m)$ *and an alignment factor* $\phi_{\mathtt{I_1}} \in \text{ALIGN\_FACTOR}(o(\mathtt{A[0]}) + \mathtt{c_1} * \mathtt{I_1} + \lambda, \mathtt{c_2}, k, m)$.

The proof follows directly from Theorems 1 and 2. The resulting loop nest is presented below:

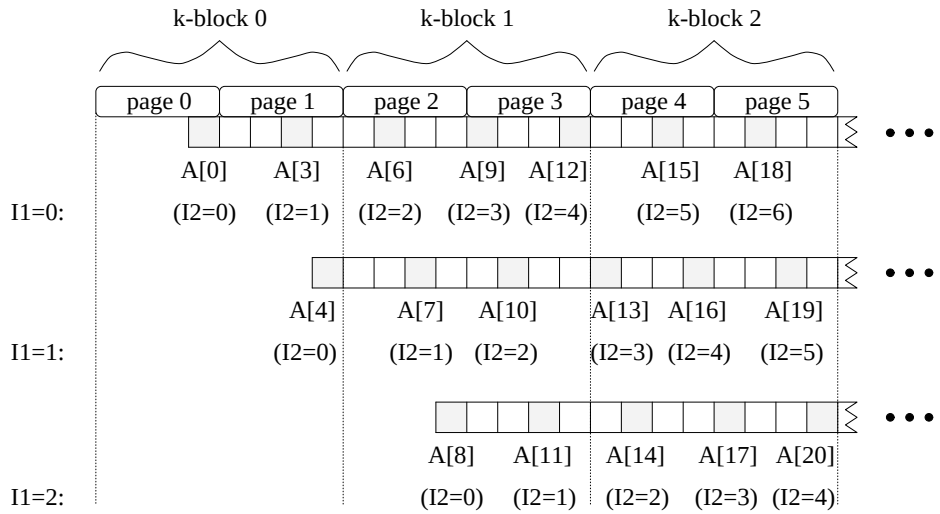


Figure 3: Example of preventing false sharing within Loop Nest 9 when R is a reference to $\mathtt{A}[4 * \mathtt{I}_1 + 3 * \mathtt{I}_2]$, and $o(\mathtt{A}[0]) = 3$, $m = 4$, $k = 2$, and $n = 0$. This example also meets Condition 1.

Loop Nest 9

```
   DO I1 = 0 TO N1-1
      φ = φ_I1
      DOALL II2 = 0 TO ⌈(N2 + φ)/β⌉-1
         DO I2 = MAX(⌈II2 * β − φ⌉, 0) TO MIN(⌈(II2 + 1) * β − φ⌉-1, N2-1)
R:          A[c1 * I1 + c2 * I2 + λ] = h(I1, I2)
         END DO
      END DOALL
   END DO
```

Example 6 Assume that Loop Nest 9 contains the reference R to $\mathtt{A}[4 * \mathtt{I}_1 + 3 * \mathtt{I}_2]$. Let $m = 4$ and $o(\mathtt{A}[0]) = 1$. We arbitrarily choose $k = 2$ and $n = 0$, so that the blocking factor is $\beta(2) = 8/3$ and the alignment factor is $\phi(0) = ((3 + 4 * \mathtt{I}_1) \bmod 8)/3$. The mapping between pages and DOALL loop iterations that results when executing Loop Nest 9 under these conditions is displayed in Figure 3. Note that, although we prevented page-level sharing within a single DOALL loop execution, we have not necessarily prevented it across DOALL loop executions.

For example, `A[9:12]` lie on the same page. Because `A[9]` is accessed during the same DOALL loop execution as `A[12]`, they will be both be accessed by the same processor. However, `A[9]` and `A[11]` are accessed during distinct DOALL loop executions. Therefore they may be accessed by different processors resulting in page-level sharing. Preventing this latter type of sharing is the subject of Section 4.2.

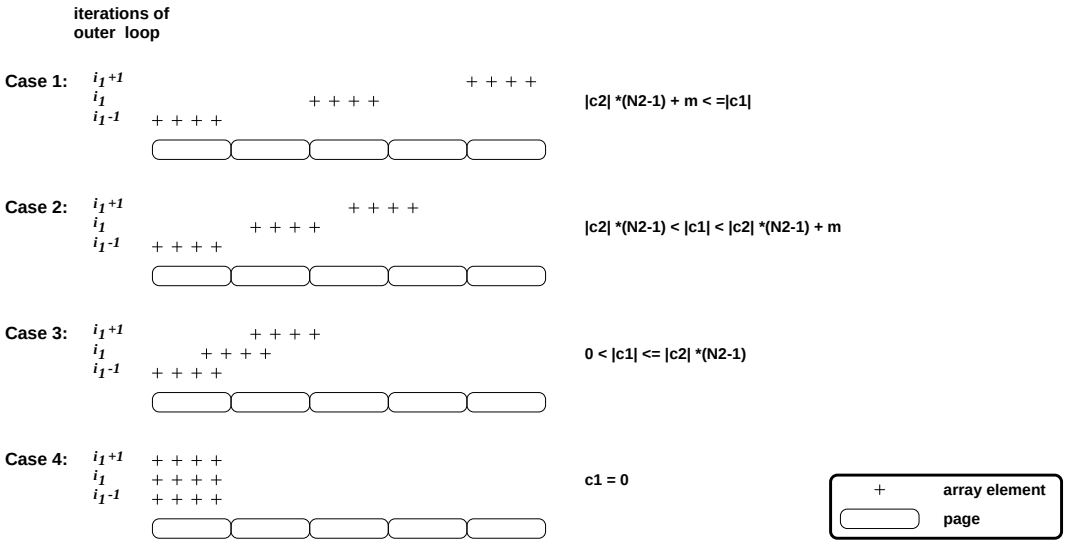


Figure 4: Four cases for the footprint of the reference to $\mathtt{A}[\mathtt{c}_1 * \mathtt{I}_1 + \mathtt{c}_2 * \mathtt{I}_2 + \lambda]$.

4.2 Step 2: Preventing Page-level Sharing across Doall Loop Executions

Consider the memory reference pattern that arises from the reference R during the execution of Loop Nest 9. Within a single execution of this DOALL loop, footprint of R is a range of length $\mathtt{c}_2 * (\mathtt{N}_2 - 1) + 1$ and a stride of $\mathtt{c}_2$. This footprint moves across $\mathtt{A}$ at a rate of $\mathtt{c}_1$ locations per iteration of the $\mathtt{I}_1$-loop.

When analyzing the page level sharing generated by $\mathtt{A}$, four cases can arise. These are depicted in Figure 4.

Case 1: $|\mathtt{c}_2| * (\mathtt{N}_2 - 1) + m \leq |\mathtt{c}_1|$.

Each page is accessed during *at most* one DOALL loop execution, so there is no page-level sharing across DOALL loop executions. Consequently, if β and ϕ meet the requirements of Theorem 3, then all page-level sharing is prevented.

Cases 2–4: $0 \leq |\mathtt{c}_1| < |\mathtt{c}_2| * (\mathtt{N}_2 - 1) + m$.

The same page might be accessed during two or more executions of the DOALL loop. In this case, the goal is to distribute pages amongst processors so that the processor that executes the iterations associated with a given page during one execution of the DOALL loop executes the iterations associated with that page during any other executions of the DOALL loop in which that page is accessed.

Before this approach can be presented, the notion of a *consistent partitioning* is needed.

Definition 5 *The partitioning of pages into k-blocks is* **consistent** *if and only if the same partitioning is used during every DOALL loop execution.*

Example 7 The partitioning in Figure 3 is consistent.

Theorem 4 *Assume that a block size* $\beta \in$ BLK_SZ($\mathtt{c_2}, m$) *and an alignment factor of* $\phi_{\mathtt{I_1}} \in$ ALIGN_FACTOR($o(\mathtt{A[0]}) + \mathtt{c_1} * \mathtt{I_1} + \lambda, \mathtt{c_2}, k, m$) *are selected. Page-level sharing between processors can be prevented altogether with respect to* R *if the following two conditions are both met:*

Condition 1: *the partitioning of pages into* k*-blocks is consistent across DOALL loop executions, and*

Condition 2: *if a* k*-block is accessed by processor* j *during one DOALL loop execution and by processor* j' *during another DOALL loop execution, then necessarily* $j = j'$*.*

Proof Assume that Condition 1 is met. Then pages are partitioned into k-blocks that are treated as indivisible "pages" of size $k * m$ and this partitioning is the same across every execution of the DOALL loop. During any given DOALL loop execution, only one processor can execute iterations that map to pages in that block. Therefore, page-level sharing between processors can occur only if the k-block that is accessed by one processor during one DOALL loop execution is accessed by a different processor during another DOALL loop execution. This violates Condition 2. □

In the remainder of this section, we show how Conditions 1 and 2 can be met in Cases 2–4 from Figure 4.

4.2.1 Meeting Condition 1

When $|\mathtt{c_2}| \geq m$, there is no page-level sharing within a single DOALL loop execution. However, in Cases 2–4, there may be page-level sharing across DOALL loop executions. Therefore, to meet Condition 1, it is necessary to assign iterations in blocks of pages, even when there is no page-level sharing within one execution of the DOALL loop (i.e., even when $|\mathtt{c_2}| \geq m$).

Definition 6 ϕ *is a* **consistent alignment factor** *if* ϕ *is an alignment factor and a consistent partition is effected in Loop Nest 9 when* ϕ *is used in combination with a blocking factor of* $\beta = \beta(k), k \in \mathbf{P}$*.*

Theorem 5 *Assume that* $\beta = \beta(k)$*,* $k \in \mathbf{P}$*. If* n *is invariant with respect to the* $\mathtt{I_1}$*-loop, then* $\phi_{I_1}(n)$ *is a consistent alignment factor.*

Proof The value of n determines the partitioning of pages into k-blocks. If n is invariant with respect to the $\mathtt{I_1}$-loop, then the same partitioning of pages into k-blocks is used during every iteration of the $\mathtt{I_1}$-loop. □

4.2.2 Meeting Condition 2

To meet Condition 2, we must ensure that every time a k-block is accessed, it is accessed by the same processor. Within a DOALL loop execution this is ensured once a consistent partitioning is effected, because

each k-block is treated as an indivisible unit, and during each iteration of the $\mathtt{II_2}$ loop iteration a single k-block is accessed.

To ensure this across iterations of the outer loop, use a page-based owner computes rule to partition iterations. First, we define a surjective mapping π from k-blocks to processors. Then, we ensure that, during each $\mathtt{I_1}$-loop iteration, any $\mathtt{II_2}$-loop iterations that access k-block p_k are assigned to processor $\pi(p_k)$. The mapping that we choose is

$$\pi(p_k) = p_k \bmod P,$$

where P is the number of processors.

Example 8 Consider the reference $\mathtt{A}[\mathtt{4} * \mathtt{I_1} + \mathtt{3} * \mathtt{I_2}]$ again, under the same conditions as in Example 6. Figure 5 shows the assignment of k page blocks to processors that results when the above mapping is used. Note that page-level sharing between processors is now prevented.

We now derive the transformation needed to effect this mapping.

Lemma 7 *Suppose that k-block p_k^n is accessed during iteration $ii_2 \in \mathtt{II_2}(\mathtt{I_1} = i_1)$ and that k-block p_k^n maps to processor P_{pid}. Then the complete set of k-blocks accessed during iteration $\mathtt{I_1} = i_1$ that map to processor pid are*

$$\{\ p_k^n + t * P \mid ii_2 + sign(\mathtt{c_2}) * t * P \in \mathtt{II_2}(\mathtt{I_1} = i_1),\ t \in \mathbb{Z}\ \}.$$

Proof The set of *k-block*s that map to processor pid is $\{\ \pi(p_k^n + t * P) \mid t \in \mathbb{Z}\ \}$. During any given iteration of the $\mathtt{II}_2$-loop, a single k-block is accessed. Suppose that $ii_2, ii_2 + sign(\mathtt{c_2}) * t * P \in \mathtt{II_2}(\mathtt{I_1} = i_1)$, $t \in \mathbb{Z}$. If k-block p_k^n is accessed during iteration ii_2, then *k-block* $p_k^n + t * P$ is accessed during iteration $ii_2 + sign(\mathtt{c_2}) * t * P$. □

Lemma 8 *The lowest numbered iteration $ii_2 \in \mathtt{II_2}(\mathtt{I_1} = i_1)$ that maps to processor pid is $ii_2 = (sign(\mathtt{c_2})\,(pid - p_k^n(\mathtt{A}[\mathtt{c_1} * \mathtt{I_1} + \lambda])) \bmod P$.*

Proof Suppose that iteration $\mathtt{II_2} = ii_2$ is the first iteration executed by processor pid. k-block $p_k^n(\mathtt{A}[\mathtt{c_1} * \mathtt{I_1} + \lambda] + sign(\mathtt{c_2}) * ii_2$ is accessed during iteration ii_2. Based on the chosen mapping function, ii_2 must be the smallest positive integer such that $(p_k^n(\mathtt{A}[\mathtt{c_1} * \mathtt{I_1} + \lambda] + sign(\mathtt{c_2}) * ii_2) \bmod P = pid$. Solving for the smallest $ii_2 \in \mathbb{N}$ completes the proof. □

The transformed loop nest is shown below:

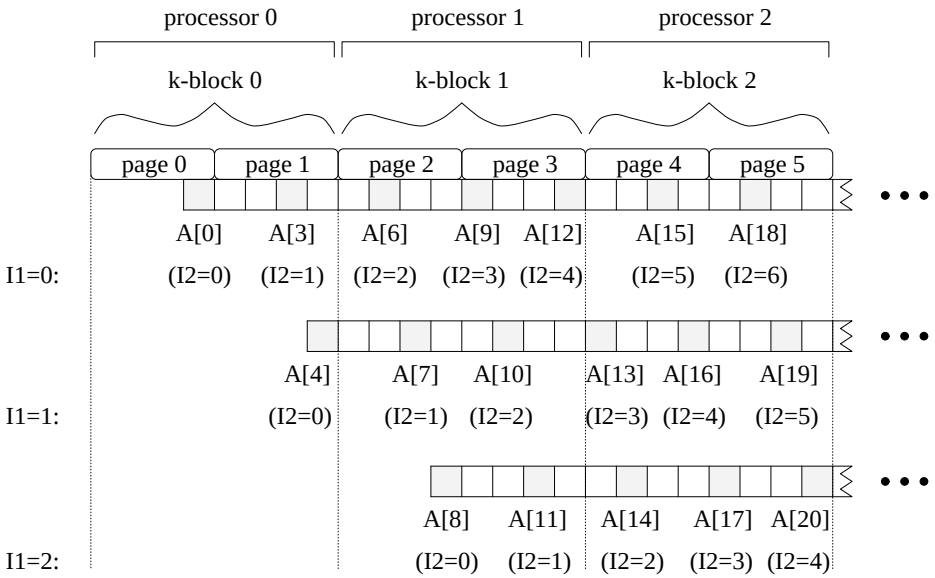


Figure 5: Example of meeting Conditions 1 and 2. R is a reference to $\mathtt{A}[4 * \mathtt{I}_1 + 3 * \mathtt{I}_2]$ in Loop Nest 10. $o(\mathtt{A}[0]) = 3$, $m = 4$, $k = 2$, and $n = 0$.

Loop Nest 10

```
    β = β(k)
    DO I1 = 0 TO N1-1
       φ = φ_I1(n)
       DOALL II2'=0 TO P-1
          pid = GetPid()
          firstIter = (sign(c2(pid − p_k^n(A[c1 I1 + λ])) mod P
          DO II2= firstIter TO ⌈(N2 + φ)/β⌉ − 1 by P
             DO I2 = MAX(⌈II2 * β − φ⌉, 0) TO MIN(⌈(II2 + 1) * β − φ⌉-1, N2-1)
R :             A[c1 * I1 + c2 * I2 + λ] = h(I1, I2)
             END DO
          END DO
       END DOALL
    END DO
```

P is the number of processors allocated to the loop nest. The function $GetPid()$ returns an identifier between 0 and $P - 1$ that represents the identifier of the processor to which a DOALL loop iteration is mapped. If a DOALL loop with P iterations is executed, it is also assumed that each distinct DOALL loop iteration is mapped to a distinct processor. Because there are at most $\lceil N_2/\beta \rceil + 1$ k-blocks accessed during any DOALL loop execution, at most $\lceil N_2/\beta \rceil + 1$ processors can be productively used.

Example 9 Figure 5 shows the mapping between k-blocks and processors that results when Loop Nest 10 is executed using the same reference and parameters as in Example 6. Note that a given *k-block* is always mapped to the same processor. Therefore, both Conditions 1 and 2 are now met.

Lemma 9 *Loop nest 10 is equivalent to Loop Nest 9.*

Proof Loop Nest 10 is equivalent to Loop Nest 9 if there is a one-to-one correspondence between the values that $\mathtt{II_2}$ takes on during execution of Loop Nest 10 and the set of integers $\{0, 1, \ldots, \lceil(\mathtt{N_2} + \phi)/\beta\rceil - 1\}$. Let $t \in \mathbb{Z}$. During a given iteration of the $\mathtt{I_1}$-loop in Loop Nest 10, processor *pid* executes $\mathtt{II_2}$-loop iterations

$$\mathcal{S}(pid) = \{\ \mathtt{firstIter}(pid) + t * P\ \} \cap [0 : \lceil(\mathtt{N_2} + \phi)/\beta\rceil - 1]_{\mathbb{Z}},$$

where $\mathtt{firstIter}(pid) = (sign(\mathtt{c_2})\,(pid - p_k^n(\mathtt{A}[\mathtt{c_1} * \mathtt{I_1} + \lambda]))) \bmod P$. Note that $\bigcup_{pid=0}^{P-1} \mathtt{firstIter}(pid) = \{\ 0, 1, \ldots, P-1\ \}$. Therefore, the set of values that $\mathtt{II_2}$ takes on during execution of $\mathtt{I_1}$-loop, is

$$\bigcup_{pid=0}^{P-1} \mathcal{S}(pid) = \{0, 1, \ldots, \lceil(\mathtt{N_2} + \phi)/\beta\rceil - 1\}.$$

Trivially, no $\mathtt{II_2}$-loop iteration is executed more than once within any given execution of the $\mathtt{II}$-loop. Moreover, no processor executed the same $|ttII_2$ loop iteration more than once and, given two processors *pid* and *pid'*, $\mathcal{S}(pid) \cap \mathcal{S}(pid') = \emptyset$. Thus, a one-to-one correspondence is established. □

Theorem 6 *Loop Nest 10 meets Conditions 1 and 2.*

The proof follows directly from Lemmas 7–9.

Note that the scheduling of DOALL loop iterations in Loop Nest 10 is static. Alternatively, a consistent surjective mapping could be effected while allowing DOALL loop iterations to be dynamically scheduled [GW92].

5 Parallelizing the Outer Loop

Suppose instead that we parallelize the outer loop. In this case, each execution of the $\mathtt{I_2}$-loop (equivalently, each iteration of the original $\mathtt{I_1}$-loop) is now indivisible and hence must be executed in its entirety, with the iterations executed in the existing order. Because the outer loop is parallelized, we only need to worry about page migrations within a single DOALL loop execution (i.e., those that arise from false sharing).

Consider again the four cases for the footprint of $\mathtt{A}$, as shown in Figure 4. For simplicity, assume that array $\mathtt{A}$ is aligned with the beginning of a page boundary. The situation where this does not hold can be compensated for using a similar approach to that presented in the previous section.

Case 1: $|\mathtt{c_2}| * (\mathtt{N_2} - 1) + m \leq |\mathtt{c_1}|$

No page is accessed during more than one iteration of the DOALL loop (the $\mathtt{I_1}$-loop), so there is no false sharing.

Case 2: $|\mathtt{c_2}| * (\mathtt{N_2} - 1) < |\mathtt{c_1}| < |\mathtt{c_2}| * (\mathtt{N_2} - 1) + m$

We first consider the general case, and then discuss a simpler technique for the special case where $\mathtt{c_2} = 0$. In general, during some pairs of consecutive iterations, the sets of accessed pages overlap (e.g., iterations $\mathtt{I_1} = i_1$ and $\mathtt{I_1} = i_1 + 1$ in Case 2 of Figure 4). During other pairs of consecutive iterations, there is no overlap between the sets of pages accessed (e.g., iterations $\mathtt{I_1} = i_1 - 1$ and $\mathtt{I_1} = i_1$ in Case 2 of Figure 4).

Observation 3 *When the latter situation occurs, we have found a block boundary: iteration $i_1 - 1$ is the end of one minimum-size block and iteration i_1 is the start of the next block.*

The term *minimum* refers to the smallest set of iterations that must be scheduled as an indivisible unit to avoid false sharing.

We now derive a technique for partitioning loop iterations into minimum-size blocks. Let

$$\begin{aligned} w &= sign^{+}(\mathtt{c_1}) * \mathtt{c_1} + sign^{-}(\mathtt{c_2}) * \mathtt{c_2} * (\mathtt{N_2} - 1) + \lambda \\ z &= sign^{-}(\mathtt{c_1}) * \mathtt{c_1} + sign^{+}(\mathtt{c_2}) * \mathtt{c_2} * (\mathtt{N_2} - 1) + \lambda \ . \end{aligned}$$

Lemma 10 *No pages are shared between iterations $\mathtt{I_1} = i_1 - 1$ and $\mathtt{I_1} = i_1$ if and only if $(\mathtt{c_1} * i_1 - \mathtt{c_1} + w) \bmod m \leq w - z - 1$. ,*

Proof Let $\mathtt{A}[sm(i_1)]$ be the smallest element of $\mathtt{A}$ referenced during iteration $\mathtt{I_1} = i_1$. Let $\mathtt{A}[lg(i_1)]$ be the largest element of $\mathtt{A}$ referenced during iteration i_1. Then,

$$\begin{aligned} sm(i_1) &= \mathtt{c_1} * i_1 + sign^{-}(\mathtt{c_1}) * \mathtt{c_1} + w \\ lg(i_1) &= \mathtt{c_1} * i_1 + sign^{-}(\mathtt{c_1}) * \mathtt{c_1} + z \ . \end{aligned}$$

No pages are shared between iterations $i_1 - 1$ and i_1 if and only if there is a multiple of the page size m between the set of elements accessed during iteration $i_1 - 1$ and iteration i_1. More formally, this constraint can be expressed as

$$\left\lceil \frac{sign^{+}(\mathtt{c_1}) * lg(i_1 - 1) + sign^{-}(\mathtt{c_1}) * lg(i_1)}{m} \right\rceil < \left\lfloor \frac{sign^{+}(\mathtt{c_1}) * sm(i_1) + sign^{-}(\mathtt{c_1}) * sm(i_1 - 1)}{m} \right\rfloor \ ,$$

or equivalently, as $(\mathtt{c_1} * i_1 - \mathtt{c_1} + w) \bmod m \leq w - z - 1$. □

Let $d = \gcd(\mathtt{c_1}, m)$, and $t = m/d$. Let (u, v) be any solution to the Diophantine equation $u\,\mathtt{c_1} + v\,m = d$. (This equation, which is known as Bezout's equation, can be solved while computing $\gcd(\mathtt{c_1}, m)$ using the Euclidean algorithm. Because $d = \gcd(\mathtt{c_1}, m)$, a solution is guaranteed to exist.) Let $x = (\mathtt{j} * u - \lfloor \frac{w}{d} \rfloor * u) \bmod t$.

Theorem 7 *A block boundary occurs between iterations* $\mathtt{I_1} = i_1 - 1$ *and* $\mathtt{I_1} = i_1$ *if and only if there exist* $\mathtt{j} \in [0 : \lfloor \frac{w}{d} \rfloor - \lceil \frac{z}{d} \rceil]_{\mathbb{N}}$ *and* $\mathtt{l} \in \mathbf{Z}$ *such that* $i_1 = \text{BLK\_START}(\mathtt{j}, \mathtt{l}) = x + \mathtt{l} * t + 1$.

Proof For any $i_1 \in \mathbb{Z}$, $\mathtt{c_1} * i_1 + w \in \{\, \mathtt{j} * d + w \bmod d \,|\, \mathtt{j} \in \mathbb{N}, 0 \le \mathtt{j} \le t - 1 \,\}$. Therefore, $i_1 = \text{BLK\_START}(\mathtt{j}, \mathtt{l})$. Based on Lemma 10, if $i_1 \in I_1$ and i_1 ends a minimum-size block, then $j * d + w \bmod d \le w - z - 1$ which implies that $j \le \lfloor \frac{w}{d} \rfloor - \lceil \frac{z}{d} \rceil$. □

Based on this result, we can compute the corresponding end to this minimum-size block

$$\text{BLK\_END}(\mathtt{j}, \mathtt{l}) = \min(\ \text{BLK\_START}(\mathtt{j}+1, \mathtt{l}),\ \text{BLK\_START}(\mathtt{j}, \mathtt{l}+1)\ )$$

and prevent false sharing by scheduling the code as shown in Loop Nest 11. The bounds of the `j`-loop and the `l`-loop are computed to ensure that $\mathtt{I_1}$ takes on each value from the original range of the loop exactly once.

Loop Nest 11

```
   DOALL j = JMIN TO JMAX
      DOALL l = LMIN(j) TO LMAX(j)
         /* I1-loop iterates over the next block */
         DO I1 = MAX(0,BLK_START(j,l))
               TO MIN(BLK_END(j,l),N1 - 1)
            DO I2=0 TO N2-1
R :            A[c1 * I1 + c2 * I2 + λ]
            END DO
         END DO
      END DOALL
   END DOALL
```

Loop Nest 12

```
   DOALL l = LMIN(0) TO LMAX(0)
      /* I1-loop iterates over the next block */
      DO I1 = MAX(0,BLK_START(0,l))
            TO MIN(BLK_END(0,l),N1 - 1)
         DO I2 = 0 TO N2-1
R :         A[c1 * I1 + c2 * I2 + λ]
         END DO
      END DO
   END DOALL
```

where $\mathtt{JMIN} = 0$ $\quad \mathtt{JMAX} = \min(t-1, \lfloor \frac{w}{d} \rfloor - \lceil \frac{z}{d} \rceil)$

$\mathtt{LMIN(j)} = \lfloor (-1 - x)/t \rfloor$ $\quad \mathtt{LMAX(j)} = \lceil (\mathtt{N_1} - x)/t \rceil - 1$.

The problem with this approach is that block sizes are very irregular and the overhead in computing the bounds can be high. Alternatively, because the reference pattern is periodic with respect to page boundaries (it repeats every t iterations), we could consider only those boundaries that occur for some fixed value of `j`, say $\mathtt{j} = 0$. This would yield constant-size blocks of t iterations each, except for any partial first and last blocks. Moreover, the computation of the relevant block boundaries is simplified. The code that results is shown as Loop Nest 12.

Example 10 Consider the reference $\mathtt{A}[\mathtt{c_1} * \mathtt{i_1} + \mathtt{i_2}]$. Assume that the page size $m = 4096$. When $\mathtt{c_1} = 512$, $\text{BLK\_START}(0, \mathtt{l}) = 7 + 8 * \mathtt{l}$. In the steady state, each block is composed of 8 iterations. However, when $\mathtt{c_1} = 513$, $\text{BLK\_START}(0, \mathtt{l}) = 4095 + 4096 * \mathtt{l}$. In this case, in the steady state, each block is composed of 4096 iterations, which may be prohibitively large.

If the block size, t iterations, is prohibitively large, array padding can be used to reduce t. This option is addressed in [Mon95].

In the special case where $\mathtt{c_2} = 0$, R is invariant with respect to the inner loop. Therefore, the simpler technique that was applied in Section 3 can be applied to the outer loop to prevent false sharing as shown in Loop Nest 13, using any block size $\beta \in \text{BLK\_SZ}(\mathtt{c_1}, m)$ and any alignment factor $\phi \in \text{ALIGN\_FACTOR}(o(\mathtt{A[0]}) + \lambda, \mathtt{c_1}, k, m)$.

Loop Nest 13

```
        DOALL II1= 0 to ⌈(N1 + φ)/β⌉-1
           DO I1 = MAX(⌈II1 * β − φ⌉, 0) TO MIN(⌈(II1 + 1) * β − φ⌉, N1)-1
              DO I2 = 0 to N2-1
R :              A[c1 * I1 + λ] = h(I1, I2)
              END DO
           END DO
        END DOALL
```

Case 3: $0 < |\mathtt{c_1}| \leq |\mathtt{c_2} * |(\mathtt{N_2} - 1)$

Finding a good compile-time technique for preventing false sharing as an optimization in this case is a topic for future research.

Case 4: $\mathtt{c_1} = 0$

Presumably this case does not occur. Otherwise, parallelization of the I_2-loop would have been illegal.

6 Preventing Page-level Sharing of Multi-Dimensional Arrays

Loop Nest 14

```
DO I1 = 0 TO N1-1
   DO I2 = 0 TO N2-1
      A[c1^0 * I1 + c2^0 * I2 + λ^0 , ..., c1^(d-1) * I1 + c2^(d-1) * I2 + λ^(d-1)]
   END DO
END DO
```

Consider the case of a two-dimensional loop nest enclosing a d-dimensional array with linear subscripts, such as that shown in Loop Nest 14. Assume that $\mathtt{c_1^0}, \ldots, \mathtt{c_1^{d-1}}, \mathtt{c_2^0}, \ldots, \mathtt{c_2^{d-1}}, \lambda^0, \ldots, \lambda^{d-1} \in \mathbb{Z}$ and that array A is stored in row-major order. Let the lower bound of dimension j be 0 and the upper bound be $D^j - 1$, $0 \leq j < \mathtt{d}$. The linearized subscript expression of the reference to A is $\mathtt{c_1}\,\mathtt{I_1} + \mathtt{c_2}\,\mathtt{I_2} + \lambda$, where $\mathtt{c_1} = \sum_{j=0}^{\mathtt{d-1}} \left(\mathtt{c_1^j} \prod_{l=j+1}^{\mathtt{d-1}} D^l\right)$, $\mathtt{c_2} = \sum_{j=0}^{\mathtt{d-1}} \left(\mathtt{c_2^j} \prod_{l=j+1}^{\mathtt{d-1}} D^l\right)$, and $\lambda = \sum_{j=0}^{\mathtt{d-1}} \left(\lambda^j \prod_{l=j+1}^{\mathtt{d-1}} D^l\right)$, where $\prod_{l=\mathtt{d}}^{\mathtt{d-1}} D^l = 1$. Once the linearized version is obtained, the theory from the preceding sections can be directly applied. It is important to note that, although subscripts expression are linearized as part of analysis, the array reference in the code itself need not be changed. It can remain as a multidimensional reference.

Loop Nest 15

```
      /* Key loop nest from Lawrence Livermore Kernel 18 */
      DOALL II2 = 0 TO ⌈(N − 2)/β⌉ − 1
        DO I1 = 2 TO 6
          DO I2 = MAX(⌈II2 * β⌉+2, 2) TO MIN(⌈(II2 + 1) * β⌉+2, 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
      END DOALL
```

7 Optimizing Loop Nests with Multiple Write References:

In practice, programs often contain DOALL loops with more than one static write reference. This section describes a technique for extending the aforementioned loop transformations to handle multiple write references. As an example for demonstrating this technique, we will use Loop Nest 15, a key loop nest from a two-dimensional explicit hydrodynamics code fragment known as Lawrence Livermore Kernel 18.

A *reference group* is a set of one or more write references with the same page offsets, the same array dimensions (excluding the outermost dimension), the same size elements, and the same subscript expressions (these need not be references to the same variable). A reference group has the 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. For example, in Loop Nest 15, `ZU` and `ZV` have the same size elements and the same subscript expressions. If they also have the same innermost dimension and the same offset, then they 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 when possible.

A reference group has the additional property that the set of block sizes and alignment factors that can be used to prevent false sharing is the same for each reference in the group. Therefore, we can prevent page-level sharing *within* a given reference group G, by applying our aforementioned techniques when applicable. If there is only one reference group, then page-level sharing is prevented altogether.

Suppose that there is more than one reference group. Page-level sharing can be prevented simultaneously for multiple reference groups by applying the following compound transformation:

- **Step 1:** Distribute the DOALL loop to encapsulate the references from each group in distinct DOALL

Loop Nest 16

```
/* Preventing false sharing for both write
   references from Loop Nest 15 */
DOALL II2' = 0 TO P-1
   pid = GetPid()

   /* Loop nest containing ZU after splitting
      main loop */
   β = β^ZU(k)
   DO II2 = 0 TO ⌈(N − 2)/β⌉-1 BY P
      DO I1 = 2 TO 6
         φ = φ^ZU_I1(n)

         /* All elements of ZU written during
            iterations (I1, I2MIN : I2MAX) lie
            within some k-block that is
            mapped to processor pid */
         I2MIN = MAX(⌈(II2+ FirstIter^ZU(pid, I1))
                 ∗β − φ⌉+2, 2)
         I2MAX = MIN(⌈(II2+ FirstIter^ZU(pid, I1)
                 +1) ∗ β − φ⌉+2, N)-1
         DO I2 = I2MIN TO I2MAX
R^ZU:       ZU[I2,I1] = ZU[I2,I1] + ···
         END DO
      END DO
   END DO
```

```
   /* Loop nest containing ZV after splitting
      main loop */
   β = β^ZV(k)
   DO II2'' = 0 TO ⌈(N − 2)/β⌉-1 BY P
      DO I1 = 2 TO 6
         φ = φ^ZV_I1(n)

         /* All elements of ZV written during
            iterations (I1, I2MIN : I2MAX) lie
            within some k-block that is
            mapped to processor pid */
         I2MIN = MAX(⌈(II2+ FirstIter^ZV(pid, I1))
                 ∗β − φ⌉+2, 2)
         I2MAX = MIN(⌈(II2+ FirstIter^ZV(pid, I1)
                 +1) ∗ β − φ⌉+2, N)-1
         DO I2 = I2MIN TO I2MAX
R^ZV:       ZV[I2,I1] = ZV[I2,I1] + ···
         END DO
      END DO
   END DO
END DOALL
```

loops.

- **Step 2:** Independently select block sizes and alignment factors for each reference group (equivalently, each loop nest).
- **Step 3:** Fuse the DOALL loops back together so that no additional synchronization is necessary.

Loop Nest 16 shows the result of applying this compound transformation to Loop Nest 15, under the assumption that references to ZU and ZV belong to different reference groups.

The first two steps of this compound transformation are always legal. These two steps alone are sufficient for preventing false sharing. However, unless the third step can also be applied, we are trading false sharing for "true" sharing, namely the introduction of additional synchronization. The third step is only legal when there are no fusion-preventing flow or anti dependences [BGM95]. For this work, we decided only to prevent page-level sharing when we could do so without introducing additional synchronization overhead.

When the third step is not legal, another alternative exists: we can prevent page-level sharing completely within some reference group. For the other reference groups, we can reduce the number of page migrations that page-level sharing causes by combining this approach with our ping-pong reduction transformation

described in [BGM94]. This hybrid strategy has been shown to work well in practice [BGM94].

8 Experimental Results

FS-OPT has been implemented in the Fortran-S compiler [BKP93], which generates code that runs on the iPSC/2 under the KOAN SVM [LP92]. The KOAN SVM 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-based 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.

The Fortran-S compiler generates code using static scheduling. When P processors are allocated to this code, there is an initial fork onto all P processors and a join at the end. The starts and ends of DOALL loops are replaced by P-processor barrier operations as needed. Whenever a DOALL loop is executed, iteration $\mathtt{I} = j$ of that DOALL loop is executed on processor j, where $0 \leq j < P$, which provides some affinity across DOALL loops.

The compiler generated two versions of each Fortran 77 benchmark studied: ORIG and FS-OPT. For the ORIG version, each processor was assigned a consecutive chunk of $\beta = N/P$ iterations, where N is the problem size. For the FS-OPT version, each processor was assigned a consecutive chunk of $\beta = \beta(\mathrm{k})$ (equivalently, k "pages" of) iterations, where k that was chosen to yield the block size $\beta(k)$ that was closest to N/P. For both versions, the innermost DOALL loop was parallelized. To maximize the grain of parallelism, loop interchanging was then applied when legal.

8.1 DMXPY

Loop Nest 17 depicts the Fortran kernel DMXPY from LINPACKD [DBMS79] which performs matrix–vector multiplication.

Loop Nest 17

```
        /* DMXPY */

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

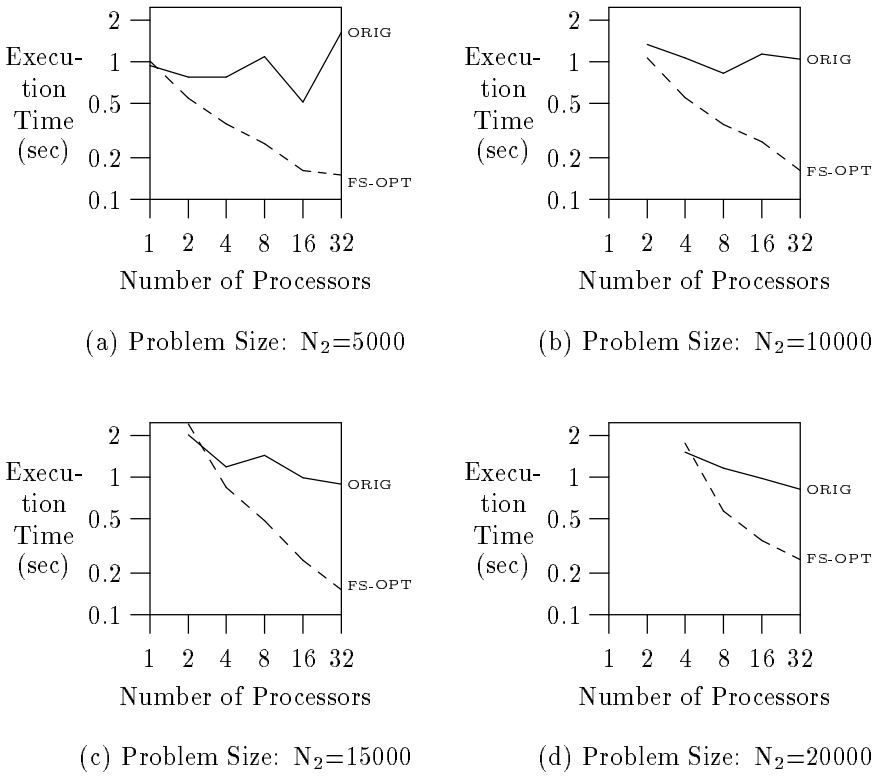


Figure 6: *Execution times for original and optimized versions of* DMXPY *(Loop Nest 17) with the inner loop parallelized and interchanged.* $\mathtt{N_1} = 10$.

Figure 6 depicts the performance of the ORIG and FS-OPT versions of these programs for four different problem sizes.

In general, the optimized version significantly outperforms ORIG. The only exception is when the number of processors is very small so that the degree of false sharing is too small to offset the load imbalance caused by FS-OPT. However, this trend quickly reverses as the number of processors is increased and the degree of false sharing with it. This effect can be seen in Figure 6(c)(d). Note that the curves that correspond to the optimized version are smoother as well. This makes the performance of the optimized versions easier to predict, which facilitates program tuning.

As can be seen in Figure 6(a), the overhead for applying FS-OPT is less than 10% of the sequential execution time. At present, no attempt is made to optimize loop overhead. We expect to reduce this number significantly further by applying standard optimizations and exploiting cases where operands and divisors are powers of two to substitute expensive operations with shifts and masks.

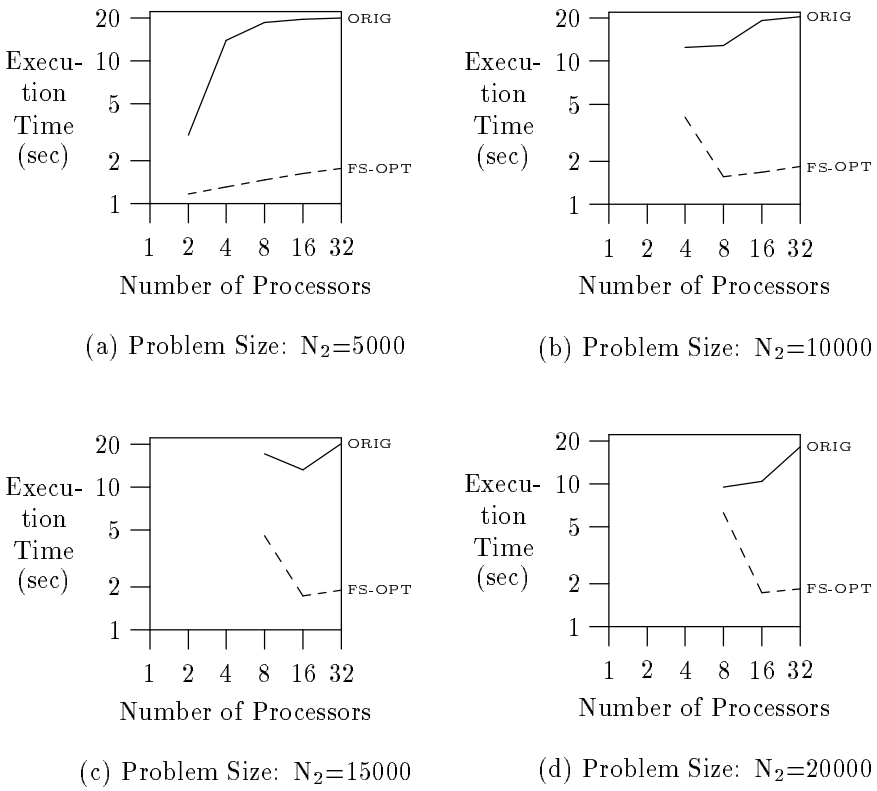


Figure 7: *Execution times for unoptimized and optimized versions of* TRIANGLE *(Loop Nest 18) with inner loop parallelized (no interchanging).* $N_1 = 100$.

8.2 TRIANGLE

Because there is processor affinity across executions of the I_1-loop in DMXPY, the reference pattern is the same on every execution of this loop, and the degree of false sharing is not very high. False sharing would become more significant if the reference pattern changed across executions of the I_1-loop. To study this case, we created the artificial benchmark TRIANGLE shown as Loop Nest 18.

Loop Nest 18

```
      /* 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 7. Again, the FS-OPT version outperforms the ORIG version. In this case, because of the triangulation, the ORIG version loses any reuse across DOALL loop executions. One beneficial side effect of the FS-OPT optimization is that it can exploit affinity across

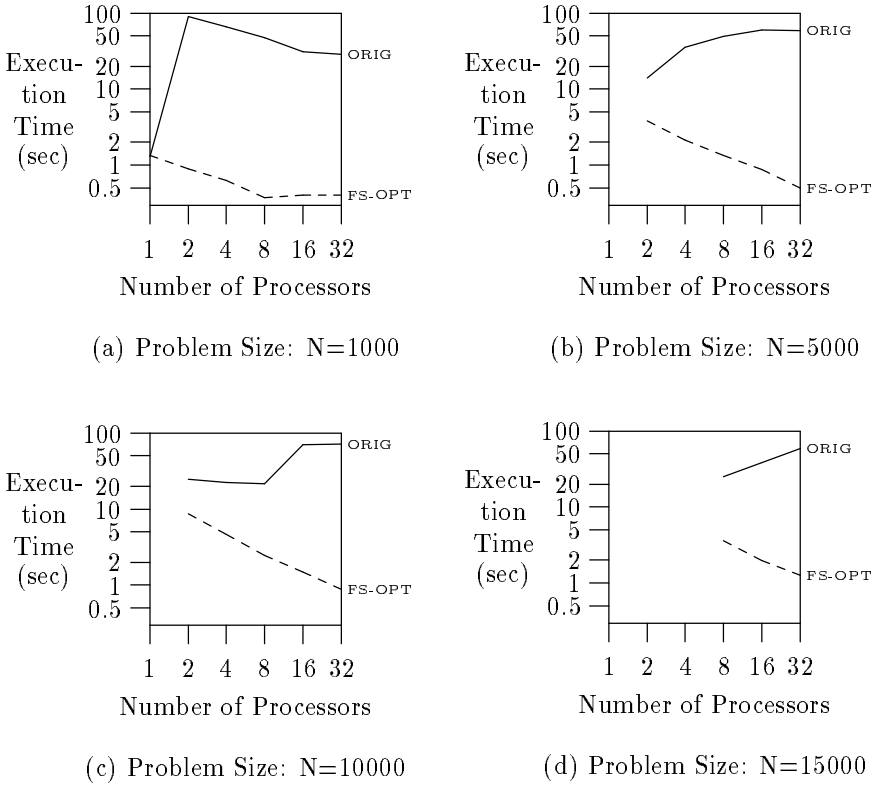


Figure 8: *Execution times for original and optimized versions of* LLK18, *with inner loops parallelized and interchanged. All arrays are aligned with page boundaries, so there is only one reference group per loop nest.*

loops.

8.3 LLK18

Figure 8 presents execution times for unoptimized and optimized versions of LLK18, a two-dimensional explicit hydrodynamics code, known as Lawrence Livermore Kernel 18. This code contains three loop nests similar to that depicted in Loop Nest 15. Although each loop nest contains multiple write references, the references within each loop nest belong to the same reference group. This is because they have the same dimensions and subscript expressions, and the Fortran-S compiler automatically aligns an array with the beginning of a page when possible. Therefore, if false sharing is prevented with respect to one write reference in each loop nest, it is automatically prevented with respect to both.

The performance of the ORIG and FS-OPT versions can be seen in Figure 8. As the number of processors increases past a threshold, the performance of FS-OPT more or less flattens out. This trend is due largely to the constraints that FS-OPT imposes 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 FS-OPT requires that pages are treated indivisibly (i.e., all writes to a given page must be performed

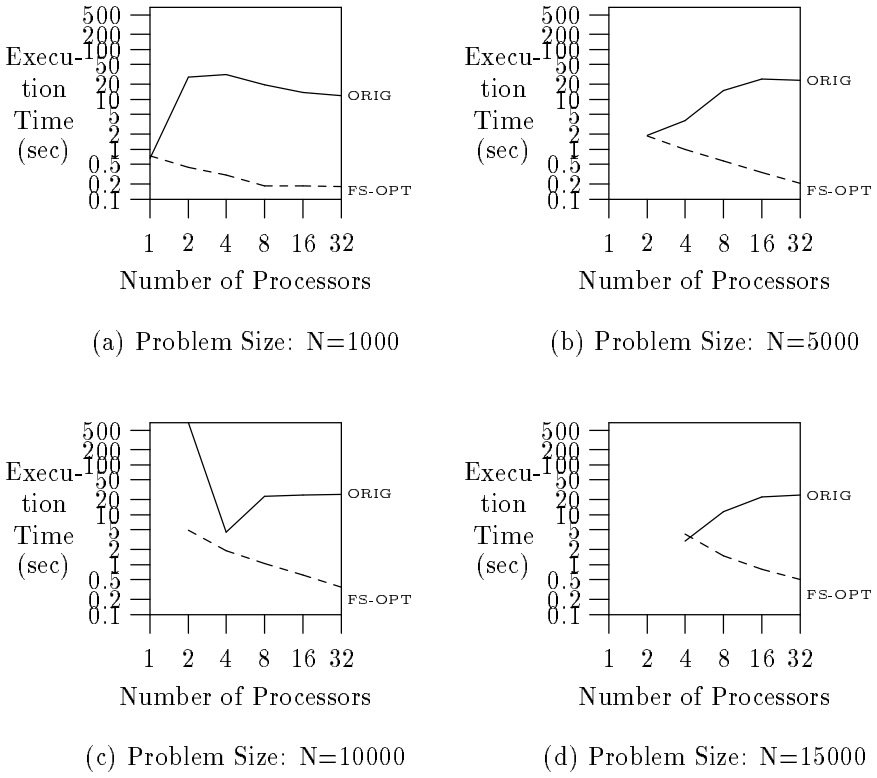


Figure 9: *Execution times for original and optimized versions of Loop Nest 15 from* LLK18*, with inner loops parallelized and interchanged.* `ZU` *and* `ZV` *are declared with different dimensions, so that Loop Nest 15 contains two reference groups.*

by the same processor), the maximum amount of parallelism is bounded from above by the number of pages. Therefore, FS-OPT has the side effect of bounding the amount of parallelism that can be exploited.

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

8.4 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 15 so that they no longer match `ZU`. The results are shown in Figure 9.

In general, for 2 to 32 processors, the optimized version greatly outperforms ORIG. One advantageous side effect of the optimizations under study is that page-level locality is increased and the working set size is decreased. Occasionally the effects are dramatic. For example, in Figure 9(c), ORIG performs very poorly on two processors because of thrashing. Because of the smaller working set of the optimized version, it performs much better.

9 Related Research

9.1 Previous Approaches to Reducing False Sharing

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 [BFS89, EJ91, LP92, TLH92, AL93, AALT94, Mon95] have been proposed.

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 optimizations that increase locality across loop nests, for example [HA90, AHD93]. Unfortunately, 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, Mon95]. 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.

Run-time solutions for preventing 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.

For this study, we targeted the prevention 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 prevented. Therefore, we did not consider any transformations that would require the insertion of additional synchronization. In contrast, on systems where no hardware or run-time support for coherence is provided, false sharing *must* be prevented to ensure correctness. Breternitz et al. [BLSS93] study this problem. They insert additional

synchronization as needed so that the compiler can maintain coherence. Consequently, their techniques are more general than ours. However, on page-coherent systems, the overhead for the additional synchronization may outweigh the performance gain from preventing false sharing.

False sharing prevention is only one optimization that a good SVM compiler should perform. Techniques to reduce synchronization and hide memory access latencies are also needed [AALT94, AHD93, MHS94, OKB, BGM95].

9.2 Relation to Research On Compiling Data Parallel Languages

Data parallel languages like HPF [KLS+94] typically support a block-cyclic data distribution for regularly distributed data. When the programmer that selects a block-cyclic distribution, data is physically distributed across processors in a block-cyclic fashion. The processor where data is physically located is the *owner* of that data. When a data-parallel program is compiled, computation is typically partitioned according to an *owner computes rule*: the processor that owns the data on the left side of statement performs the computation.

The problem of partitioning computation according to a general block-cyclic owner computes rule has been studied by several groups of researchers [CGL+93, ACIK93, KNS94, AFMP95]. The solution that we propose transforms the problem of eliminating false sharing into a similar problem: we assign pages to processors in a block-cyclic fashion and then partition computation accordingly. However, there are several significant differences that arise primarily from our goal of targeting SVM systems and from supporting a sequential language, rather than a data parallel one.

Recall that a key advantage of SVM is that it is easier to build compilers. One of our goals is to improve the quality of SVM compilers with minimal complexity increase. Therefore, our programming model differs from the traditional data-parallel model. The model that we compile for allows one type of parallel loop, a DOALL loop. This has the advantage of simplifying compiler design overall, but introduces unique problems such as those described in Section 5.

Additionally, because our input language assumes that data is laid out sequentially in memory and our target architecture is an SVM system, we have chosen to preserve the original data layout, and simply transform loops (when possible) to match the existing layout. This allows us to exploit the services that the SVM provides in transparently translating from global to local address spaces and that of automatically moving pages close to processors that are accessing them. Thus, we avoid the complexities associated with implementing global to local address translation at the compiler level which can significantly complicate the design of data parallel compilers [THK93]. Interestingly, our decision to leave the data layout intact also gives us this same advantage over most other research on compiling (either data-parallel or sequential language programs) for global address space architectures [AL93, AALT94, MHS94].

Leaving the data layout intact has the additional advantage that data is mapped directly to memory, which is linear. Thus, we only have to deal with straightforward, one-dimensional distributions. Com-

pilers for data-parallel languages such as HPF must support multi-dimensional distributions, as well as complicated alignment specifications and processor mappings. This leads to the "reaching distribution" problem [HHKT92]. In general, a compiler must know what distribution an array might have. This is particularly problematic at procedure call boundaries or when redistribution statements are present, both of which can lead to situations where variables are associated with different distributions at different points of execution. In our case, the problem is significantly simpler because there is only one type of distribution, the alignment is specified by one parameter, and there is only one mapping of blocks to processors.

10 Conclusions

The purpose of this paper has been to present a loop transformation theory that deals with the prevention of multiple-writer page-level sharing. Most importantly, we have developed a set of constraints on blocking factors, alignment factors and iteration scheduling techniques that, when met, prevent such sharing between processors. This is accomplished by partitioning the iterations into blocks of "pages" and assigns these blocks to processors as indivisible units, thus ensuring that no page is accessed by more than one processor.

In general run-time overhead is minimal. Computation can be done symbolically at compile time if necessary. Triangular loops and loops with non-unit strides can be handled. Many commonly occurring cases of loops containing multiple references can also be handled. We are currently working on techniques to generalize these optimizations even further.

Our transformation has several beneficial side effects. First, applying our transformation may expose opportunities to eliminate barrier synchronization between loops [BGM95]. Second, the transformation generally increases locality and reduces the working set of pages, thereby reducing thrashing as well. Although the derivation of the block sizes and 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. Third, performance generally becomes more predictable, which facilitates both manual and automatic program tuning.

Intuitively, the larger the page size is, the greater the degree of false sharing. Consequently, we expect the performance results obtained under the KOAN SVM system to be realizable under other SVM 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 systems such as the Kendall Square Research KSR1 and KSR2 which support smaller coherency units.

Acknowledgements

We would like to thank Thierry Priol and Zakaria Lahjomri for providing access to and assistance with KOAN. We would also like to thank Chuck Koelbel for his helpful comments on this paper.

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.

[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.

[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.

[BGM95] François Bodin, Elana D. Granston, and Thierry Montaut. Page-level Affinity Scheduling for Eliminating False Sharing. In *Fifth Workshop on Compilers for Parallel Computers*, Malaga, Spain, June 1995.

[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.

[GW92] Elana D. Granston and Harry A. G. Wishoff. Managing Pages in Shared Virtual Memory Systems: Getting the Compiler into the Game. Technical Report 92-19, Computer Science Department, Leiden University, December 1992. Revised July 1993.

[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.

[HHKT92] Mary W. Hall, Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Interprocedural Compilation of Fortran D for MIMD Distributed Memory Machines. In *Supercomputing '92*, pages 522–524. IEEE Computer Society Press, November 1992.

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

[MHS94] Ravi Michandaney, Seema Hiranandani, and Ajay Sethi. Improving the Performance of DSM Systems via Compiler Involvement. In *Supercomputing '94*, 1994.

[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.*

[THK93] Chau-Wen Tseng, Seema Hiranandani, and Ken Kennedy. Preliminary Experiences with the Fortran D Compiler. In *Supercomputing '93*, pages 338–350. IEEE Computer Society Press, November 1993.

[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.

Appendix A: Determining the Conditions Under Which Integer Block Sizes and Alignment Factors Can Be Found

Consider the following loop nest containing a reference R to $\mathtt{A}[\mathtt{c} * \mathtt{I} + \lambda]$, where $\mathtt{I}$ is the index of the enclosing loop, and $\mathtt{c}, \lambda \in \mathbb{Z}$ are invariant with respect to the loop.

Loop Nest 19

```
     DOALL I = 0 to N-1
R:         A[c*I+λ] = h(I)
     END DOALL
```

Using a non-integer block size β and alignment factor ϕ generally does not add significant overhead. Moreover, for the reasons given below, the block size will often end up being an integer. If not, however, the block size can be restricted to integer values at the cost of potentially increasing the minimum block size that meets this requirement. Furthermore, whenever the block size is an integer, an integer alignment factor can be found.

Theorem 8 *If* $\mathtt{c} \mid k * m$[5] *, then*
(a) *block size* $\beta = \beta(k) \in \mathbb{Z}$*, and* (b) $\lfloor \phi(n) \rfloor \in \mathbb{Z}$ *is an alignment factor.*

Proof (a) Trivial.
(b) Let $i \in \mathtt{I}$, $ii \in \mathtt{II}$, and assume that $\mathcal{R}(\phi, \beta, ii)$ is defined as in Section 3.3. Then $\phi(n)$ effects the following mapping from $\mathtt{I}$-loop iterations to $\mathtt{II}$-loop iterations:

$$i \mapsto ii \;\Leftrightarrow\; i \in \mathtt{I}(\mathtt{II} = ii) \;\Leftrightarrow\; i \in \mathcal{R}(\phi(n), \beta, ii) \cap [0 : N)_{\mathbb{Z}}.$$

Because $\phi(n)$ is an alignment factor, this mapping ensures that false sharing is prevented. When $\beta \in \mathbb{Z}$, $\mathcal{R}(\phi(n), \beta, ii) = \mathcal{R}(\lfloor \phi(n) \rfloor, \beta, ii)$. Therefore, $\lfloor \phi(n) \rfloor$ effects the same mapping as $\phi(n)$, so $\lfloor \phi(n) \rfloor \in \mathbb{Z}$ is also an alignment factor. □

Note that Lemma 5 presents a condition that is sufficient but not necessary for ϕ to qualify as an alignment factor. Following directly from the above theorem are the sets of integer block sizes and alignment factors shown below:

$$\beta \in \text{iBLK\_SZ}(\mathtt{c}, m) = \begin{cases} \{\ \beta(k) = k\,\frac{m}{|\mathtt{c}|} \mid k \in \mathbb{P},\ \mathtt{c} \mid k * m\ \} & 0 < |\mathtt{c}| < m \\ \mathbb{P} & |\mathtt{c}| \geq m \end{cases} \quad (1)$$

$$\phi \in \text{iALIGN\_FACTOR}(o(\mathtt{A}[0]) + \lambda, \mathtt{c}, k, m) = \begin{cases} \{\ \lfloor \phi(n) \rfloor \mid n \in [0 : k)_{\mathbb{N}}\ \} & 0 < |\mathtt{c}| < m \\ [0 : \beta)_{\mathbb{Z}} & |\mathtt{c}| \geq m. \end{cases} \quad (2)$$

[5] The notation $x \mid y$ means "x divides y".

Observation 4 *The page size m is always a power of two. Often, the coeffcient* `c` *will also be a power of two. In this case, every $\beta(k)$ is already an integer. Whenever, $\beta(k)$ is an integer, for every rational alignment factor $\phi(n)$, there will also exist an integer alignment factor $\lfloor\phi(n)\rfloor$ that will effect the same mapping from* `I` *to* `II`*.*

Example 11 Consider the reference `A[2*I+1]`. Assume $m = 4$, $o($`A[0]`$) = 2$. Note that, for any $k \in \mathbb{P}$, β is an alignment factor. Arbitrarily choose $k = 2$ and $n = 0$. Then $\beta = \beta(2) = 4$. The effects of applying alignment factors $\phi = \phi(0) = 3/2$ and $\phi = \lfloor\phi(0)\rfloor = 1$, respectively, are depicted in the diagram below. Note that, because `A[6]` and `A[14]` are not accessed, the same mapping from `I`-loop iterations to `II`-loop iterations (equivalently, from array elements to `II`-loop iterations) is obtained with either alignment factor.