Evaluating Two Loop Transformations for Reducing Mutliple-Writer False Sharing

Franois Bodin
Elana D. Granston
Thierry Montaut

CRPC-TR94479
August, 1994

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

From the 7th Annual Workshop on *Languages and Compiler for Parallel Computing*, Ithaca, New York, August, 1994. Abridged version to be published by Springer-Verlag.

Evaluating Two Loop Transformations for Reducing Multiple-Writer False Sharing

François Bodin* bodin@irisa.fr
Elana D. Granston† granston@cs.rice.edu
Thierry Montaut* montaut@irisa.fr

* IRISA, Campus de Beaulieu, 35042 Rennes, Cedex, France

† Rice University, Center for Research on Parallel Computation
6100 S. Main Street, Houston, Texas 77005, USA

Abstract. To simplify the programming of hierarchical and distributed-memory parallel systems, the notion of *shared virtual memory* (SVM) has been proposed. This abstraction provides the programmer with the illusion of a flat global address space and coherence is maintained at the page level. The success of this abstraction depends on the efficiency of page management. This in turn depends on the efficiency of handling *false sharing* and the resulting *ping-pong* effects that it can cause. In this paper, we evaluate two loop transformations for attacking this problem. The first is a simple, new, compile-time technique for reducing the ping-pong effects that result from multiple-writer false sharing. The second is our previously-proposed technique for eliminating multiple-writer false sharing itself. These techniques have been implemented in the Fortran-S compiler, which generates code that runs on the iPSC/2 under the KOAN SVM. Preliminary performance results are presented.

1 Introduction

To simplify the programming of parallel systems with memory hierarchies and physically distributed address spaces, much research effort has been directed toward providing the programmer with the illusion of a global address space, known as *shared virtual memory* (SVM) [1]. SVM shields the programmer from the underlying memory architecture of a distributed-memory parallel computer by providing a virtual address space consisting of pages, where a *page* is the unit of data to which coherency is applied. (In practice, this coherency unit may be an actual physical page, a cache line, or a multiple or portion thereof.) Pages are physically distributed according to some mapping function.

The success of this SVM abstraction depends heavily on page caching and on the existence of page-level locality. Unfortunately, *false sharing* can prevent the exploitation of this locality. Especially problematic is *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 invalidate-based coherence protocol whereby, before a processor can write to a page, all other copies must be invalidated. Then multiple-writer false sharing

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
```

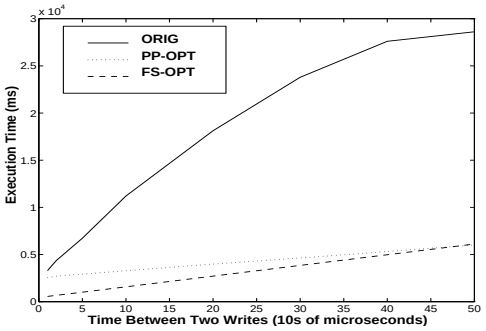

Fig. 1. *Execution times for three versions of Loop Nest 1.*

causes writes to be serialized. Furthermore, it causes a minimum of n_P-1 page faults for a given page, where n_P is the number of processors simultaneously accessing that page. To reach this minimum, each processor must be able to complete all of its accesses to that page before the next processor begins. When this is not the case, the page can bounce back and forth repeatedly between processors, causing the number of page faults to rise much higher. These additional page faults are referred to as *ping-pong* effects. The term *ping-pong* is used because the affected page bounces back and forth between processors.

As an example, consider Loop Nest 1. Execution times are shown in Fig. 1 for three versions of this loop nest:

- the original loop (ORIG),
- a version optimized to reduce ping-pong effects only (PP-OPT), and
- a version optimized to eliminate multiple-writer false sharing (FS-OPT).

All three versions were executed on 16 processors of a 32-processor iPSC/2 under the KOAN SVM system [2], which supports the aforementioned invalidate-based coherence protocol and employs a page size of 4 KB (512 double-precision numbers). (A brief overview of KOAN can be found in Appendix A.) The problem size was $\mathtt{N_1} = \mathtt{N_2} = 10^3$. The performance difference between the two optimized versions and ORIG provides a conservative approximation of the degradation that can be caused by multiple-writer false sharing alone, and by multiple-writer false sharing compounded by ping-pong effects.

In this paper, we briefly describe the two transformations used in the aforementioned experiment. We also present preliminary performance results for these transformations. The remainder of this paper is organized as follows. Sect. 2 and Sect. 3 discuss the application of PP-OPT and FS-OPT to loops containing a single static write reference. Sect. 4 extends these techniques to loops containing multiple static write references. Sect. 5 presents performance results. Sect. 6 compares our approach to related work in the area of false sharing. Sect. 7 summarizes this work and discusses future research directions.

2 Reducing Ping-pong Effects (PP-OPT)

Ping-pong effects occur only when at least one processor is writing to a page multiple times *and* there is sufficient time between successive writes by this processor for a second processor to acquire the page, thus causing this first processor to fault on a successive write. This phenomenon can be seen in Fig. 1, where the time between successive writes is varied along the x-axis. As this time increases, ping-pong effects become more pronounced. The phenomenon is due to a combination of the page-transfer protocol and the relative magnitude of the network latency in comparison to the time between writes. Although the actual values are characteristic of KOAN, these trends should generalize to other SVM systems.

Therefore, ping-pong effects can be reduced by minimizing the time between successive writes by the same processor to the same page. This can be accomplished by stripmining the parallel loop and then delaying the non-local writes within each strip until the strip's end, causing these writes to be performed in quick succession. The main advantages of this simple, new technique are its wide applicability and its simplicity of implementation at the compiler level. The result of applying this optimization to Loop Nest 1 can be seen in Loop Nest 2. The local array `buff` serves as a software write buffer which stores results locally between the time that they are computed and the time that they are written out.

Loop Nest 2

```
LOCAL ARRAY: buff[0:buffsz-1]

DO I1 = 0 TO N1-1
  DOALL II2 = 0 TO N2-1 by buffsz

    /* Main loop */
    J = 0
    DO I2 = II2 TO MIN(II2+buffsz, N2)-1
      buff[J]=h(I1,I2)
      J = J + 1
    END DO

    /* Copy-out loop: A ← buff */
    J = 0
    DO I2 = II2 TO MIN(II2+buffsz, N2)-1
      A[2*I1 + 10*I2 + 1] = buff[J]
      J = J + 1
    END DO
  END DOALL
END DO
```

As one might intuitively expect, there is a space–performance tradeoff: the larger the buffer, the better the performance of PP-OPT. For more details on this transformation itself and on the cost–benefit tradeoffs involved in selecting buffer sizes, see [3, 4].

3 Eliminating False Sharing (FS-OPT)

For cases where reducing ping-pong effects is insufficient, we review our previously proposed transformation for eliminating multiple-writer false sharing itself. In this section, we present examples of applying this optimization to loop nests containing a single static write reference. In the interest of brevity, we restrict our discussion to cases that occur in the benchmarks discussed later in this paper.

3.1 Handling One-Dimensional Loop Nests

Consider Loop Nest 3. Multiple-writer false sharing can be eliminated by partitioning pages into blocks of k pages, known as *k-blocks*. Then we can partition the computation such that during any given DOALL loop iteration, the set of `I`-loop iterations that are executed are exactly those that map to some distinct k-block.

Assume that a page can hold precisely m elements of `A` and that $o(\texttt{A[}expr\texttt{]})$ is the offset of `A[`$expr$`]` on some page, where $0 \leq o(\texttt{A[}expr\texttt{]}) < m$. The code to accomplish this partitioning is shown in Loop Nest 4.[1]

Loop Nest 3

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

Loop Nest 4

```
   DOALL II = 0 TO ⌈(N + φ)/β⌉-1
     /* I-loop iterates over exactly one k-block */
     DO I = MAX(⌈II * β − φ⌉, 0) TO
                MIN(⌈(II + 1) * β − φ⌉, N)-1
R:              A[3*I] = h(I)
     END DO
   END DOALL
```

where

$$\beta \in \{\, \beta^{\texttt{A}}(k) = k * \frac{m}{3} \mid k \in \mathbb{P} \,\}$$

$$\phi \in \left\{ \phi^{\texttt{A}}(n) = \frac{(o(\texttt{A[0]}) + n * m) \bmod (k * m)}{3} \mid n \in [0 : k)_{\mathbb{N}} \right\} .$$

Depending on the choice of k, applying this transformation may lead to the use of a non-integer *block size* β. The *alignment factor* ϕ compensates for the fact that the array element accessed during iteration `I`=0 falls in the middle of a k-block. A comprehensive discussion of this optimization can be found in [5].

As an example, let $m = 4$ and $o(\texttt{A[0]}) = 3$.[2] We arbitrarily choose $k = 2$ and $n = 0$ so that the block size is $\beta = 8/3$ and the alignment factor is $\phi = 1$. Fig. 2

[1] We use the following set notation: $\mathbb{N} = \{0, 1, \ldots\}$, $\mathbb{P} = \{1, 2, \ldots\}$, and $\mathbb{R}$ is the set of real numbers. $[mn : mx) = \{r \in \mathbb{R} \mid mn \leq r < mx\}$. $[mn : mx)_S = [mn : mx) \cap S$, where $S \in \{\mathbb{N}, \mathbb{P}\}$.

[2] For illustrative purposes, an unrealistically small page size has been chosen.

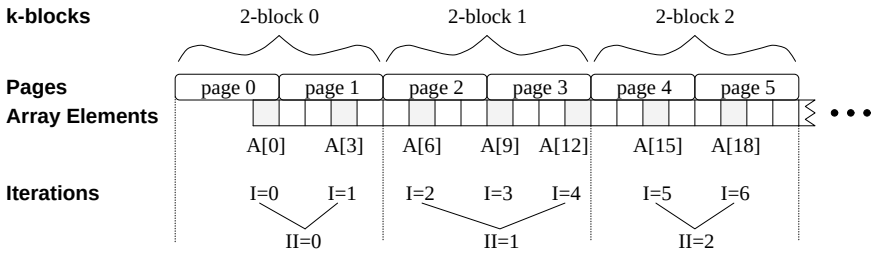

Fig. 2. *Partitioning of iterations/pages that results after executing Loop Nest 4 when* $m = 4$, $o(\texttt{A[0]}) = 3$, $\beta = 8/3$, *and* $\phi = 1$.

shows the partitioning of iterations/pages into k-blocks that results. 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 2-block.

Consider elements `A[9]` and `A[12]`, which both lie on page 3 and are accessed during iterations `I`=3 and `I`=4, respectively. To prevent false sharing of this page, both of these iterations must be executed by the same processor.[3] This is ensured in two steps. First, the iteration partitioning strategy maps both of these iterations to the same 2-block. Then, it guarantees that all iterations that map to this 2-block are executed by the same processor.

Because the blocking factor is not an integer, the number of iterations that map to a k-block can vary by one from block to block. 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.

3.2 Handling Two-Dimensional Loop Nests

Consider Loop Nest 5. In this section, we show how to eliminate multiple-writer false sharing and other sources of page migration from this example loop nest. A more general discussion of two-dimensional loop nests can be found in [4, 5].

Loop Nest 5

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

In the above loop nest, page faults due to write references could arise from one or more of the following sources:

- **Source 1:** cold start misses,
- **Source 2:** multiple-writer false sharing within a single execution of a DOALL loop, and

[3] Had we chosen a larger, more realistic page size, there would have been more cases similar to this.

– **Source 3:** overlap between the sets of pages written during two distinct executions of a DOALL loop.

In our experience, cold start misses are generally insignificant compared to those arising from the remaining two sources; therefore, we ignore these. To eliminate page migrations of the second and third sources, we apply the same technique as in the case of one-dimensional loop nests, but impose the additional restrictions that (1) the same partitioning of pages into k-page blocks must be used during every iteration of the $\mathtt{I_1}$ loop and (2) a surjective mapping must be established between k-blocks and processors that is enforced at run time, so that each processor executes exactly those iterations associated with "its" k-blocks. The mapping that we choose maps every P^{th} k-block to the same processor, where P is the number of processors. This effects a block-cyclic schedule.

The transformed code is shown in Loop Nest 6. It is assumed that if a DOALL loop with P iterations is executed, then each distinct DOALL loop iteration is mapped to a distinct processor.

Loop Nest 6

```
DO I1 = 0 TO N1-1
  DOALL II2' = 0 TO P-1
    pid = GetPid()
    DO II2 = firstIter^A(pid,I1) TO ⌈(N2 + φ/β)⌉ − 1 by P
      /* I2-loop iterates over exactly one k-block */
      DO I2 = MAX(⌈II2 * β − φ⌉, 0) TO MIN(⌈(II2 + 1) * β − φ⌉, N2)-1
R:          A[4 * I1 + 3 * I2] = h(I1, I2)
      END DO
    END DO
  END DOALL
END DO
```

where

$$\beta \in \{ \beta^{\mathtt{A}}(k) = k * \frac{m}{3} \mid k \in \mathbb{P} \}$$

$$\phi \in \left\{ \phi^{\mathtt{A}}_{\mathtt{I_1}}(n) = \frac{(o(\mathtt{A}[\mathtt{0}]) + 4 * \mathtt{I_1} + n * m) \bmod (k * m)}{3} \mid n \in [0:k)_{\mathbb{N}} \right\}$$

$$\mathit{firstIter}^{\mathtt{A}}(\mathtt{pid}, \mathtt{I_1}) = \left(\mathtt{pid} - \left\lfloor \frac{o(\mathtt{A}[\mathtt{0}]) + 4 * \mathtt{I_1} + n * m}{k * m} \right\rfloor \right) \bmod P \ .$$

As an example, assume that $m = 4$ and that $o(\mathtt{A}[\mathtt{0}]) = 3$. We arbitrarily choose $k = 2$ and $n = 0$, so that the blocking factor is $\beta = 8/3$ and the alignment factor is $\phi = ((3 + 4 * \mathtt{I_1}) \bmod 8)/3$. Fig. 3 shows the mapping between k-blocks and processors that results.

As a second example, recall Loop Nest 1. The performance benefit of applying this technique (FS-OPT) to that loop nest can be seen graphically in Fig. 1.

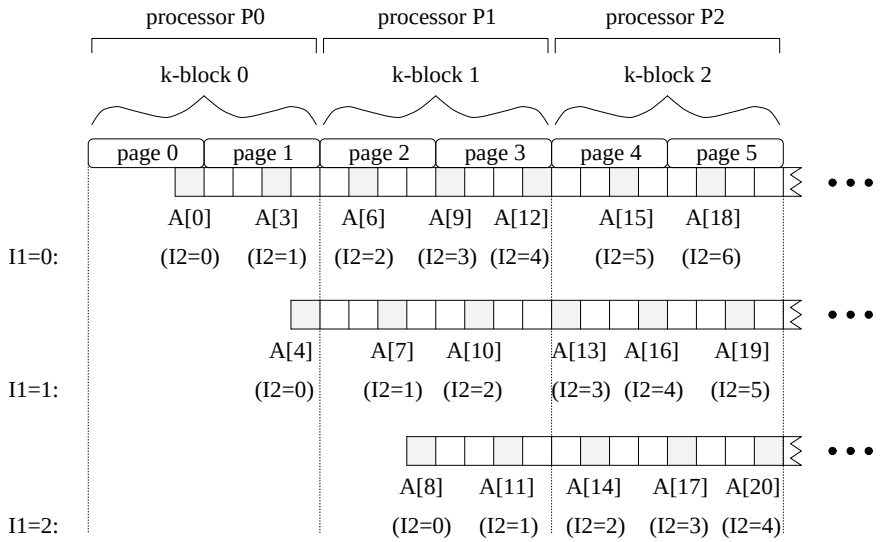

Fig. 3. *Mapping between k-blocks and processors that results after executing Loop Nest 6 when* $o(\texttt{A[0]}) = 3$, $m = 4$, $k = 2$, *and* $n = 0$.

To maximize parallelism and reduce synchronization overhead, it is generally preferable, when legal, to interchange loops so that the DOALL loop is outermost.

Note that FS-OPT eliminates page migrations across multiple executions of the same doall loop by employing the same mapping between pages and processors during each execution. This same sort of affinity can be provided across distinct loop nests that each contain write references to the same array variable by choosing the same values of k and n in each case.

4 Handling Loop Nests Containing Multiple Write References

In practice, programs often contain DOALL loops with more than one static write reference. In this section, we briefly present three new methods for extending our aforementioned strategies to handle multiple write references. These techniques differ in their aggressiveness and generality. As a running example throughout this section, we will use Loop Nest 7, a key loop nest from a two-dimensional explicit hydrodynamics code fragment known as Lawrence Livermore Kernel 18.

4.1 Technique 1: Reducing Ping-pong Effects (PP-OPT)

The technique from Sect. 2 can be readily applied to reduce ping-pong effects for both references. An example of applying this technique to Loop Nest 7 is depicted in Loop Nest 8. The effective buffer size is bounded from above by the

Loop Nest 7

```
   /* Key loop nest from Lawrence Livermore Kernel 18
      (2-D explicit hydrodynamics code fragment) */

   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
```

block size β. In Loop Nest 8, the buffer sizes are set to β. The copy-in loops, where ZU and ZV are read into their respective buffers, are not strictly necessary for this loop nest. However, we include them here because they may be needed to handle the general case and consequently are generated by default by the compiler in which these optimizations are currently implemented.

4.2 Technique 2: Eliminating False Sharing for One Reference Group and Reducing Ping-pong Effects for the Remainder (FS-PP-OPT)

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 7, 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 eliminate false sharing is the same for each reference in the group. Therefore, we can eliminate multiple-writer false sharing and Source 3 page migrations (Sect. 3.2) *within* a given reference group G, by

Loop Nest 8

```
/* Technique 1: Reducing ping-pong
   effects for both references
   from Loop Nest 7 (PP-OPT). */

LOCAL ARRAY: buffZU[0:β-1]
LOCAL ARRAY: buffZV[0:β-1]

DOALL II2 = 0 TO ⌈(N − 2)/β⌉ − 1
  DO I1 = 2 TO 6
    I2MIN = MAX(⌈II2 * β⌉+2, 2)
    I2MAX = MIN(⌈(II2 + 1) * β⌉+2, N)-1

    /* Copy-in loops:
       buffZU ← ZU, buffZV ← ZV */
    J = 0
    DO I2 = I2MIN TO I2MAX
      buffZU[J] = ZU[I2,I1]
      J = J + 1
    END DO
    J = 0
    DO I2 = I2MIN TO I2MAX
      buffZV[J] = ZV[I2,I1]
      J = J + 1
    END DO

    /* Main loop */
    J = 0
    DO I2 = I2MIN TO I2MAX
      buffZU[J] = buffZU[J] + ···
      buffZV[J] = buffZV[J] + ···
      J = J + 1
    END DO

    /* Copy-out loops:
       ZU ← buffZU, ZV ← buffZV */
    J = 0
    DO I2 = I2MIN TO I2MAX
R^ZU:   ZU[I2,I1] = buffZU[J]
      J = J + 1
    END DO
    J = 0
    DO I2 = I2MIN TO I2MAX
R^ZV:   ZV[I2,I1] = buffZV[J]
      J = J + 1
    END DO

  END DO
END DOALL
```

applying our false sharing elimination techniques from Sect. 3, when applicable. If there is only one reference group, then both of these sources of page migrations are eliminated altogether. If there is more than one reference group, then we can apply our ping-pong reduction techniques in addition to reduce ping-pong effects elsewhere.

For example, assume that `ZU` and `ZV` belong to different reference groups. Loop Nest 9 displays the result of optimizing to eliminate false sharing for the reference to `ZU` and optimizing to reduce ping-pong effects for the reference to `ZV`.

4.3 Technique 3: Eliminating Multiple-Writer False Sharing for All References (FS-OPT)

Under certain circumstances, multiple-writer false sharing and Source 3 page migrations can be eliminated 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 loops.

Loop Nest 9

```
/* Technique 2: Eliminating false
   sharing for R^ZU and reducing
   ping-pong effects for R^ZV from
   Loop Nest 7 (FS-PP-OPT). */

LOCAL ARRAY: buffZV[0:β^ZU(k)-1]

DOALL II2' = 0 TO P-1
  pid = GetPid()
  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

      /* Copy-in loop:
         buffZV ← ZV */
      J = 0
      DO I2 = I2MIN TO I2MAX
        buffZV[J] = ZV[I2,I1]
        J = J + 1
      END DO

      /* Main loop */
      J = 0
      DO I2 = I2MIN TO I2MAX
R^ZU:   ZU[I2,I1] = ZU[I2,I1] + ···
        buffZV[J] = buffZV[J] + ···
        J = J + 1
      END DO

      /* Copy-out loop:
         ZV ← buffZV */
      J = 0
      DO I2 = I2MIN TO I2MAX
R^ZV:   ZV[I2,I1] = buffZV[J]
        J = J + 1
      END DO

    END DO
  END DO
END DOALL
```

- **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.

The circumstances under which this optimization can be performed are described in [4]. An example of applying this optimization to Loop Nest 7 is depicted in Loop Nest 10. For this example, it is again assumed that the references to ZU and ZV belong to different reference groups.

5 Preliminary Experimental Results

The optimizations to reduce ping-pong effects (Sect. 2) and eliminate false sharing (Sect. 3) have been implemented in the Fortran-S compiler [6], which generates code that runs on the iPSC/2 under the KOAN SVM [2]. When P processors are allocated to this code, there is an initial fork onto all P processors and a join

Loop Nest 10

```
/* Technique 3: Eliminating false sharing for both write references
   from Loop Nest 7 (FS-OPT). */

DOALL II'_2 = 0 TO P-1
   pid = GetPid()

   /* Loop nest containing ZU after splitting main loop */
   β = β^ZU(k)
   DO II_2 = 0 TO ⌈(N − 2)/β⌉-1 by P
      DO I_1 = 2 TO 6
         φ = φ^ZU_I1(n)

         /* All elements of ZU written during iterations (I_1, I_2MIN : I_2MAX)
            lie within some k-block that is mapped to processor pid */
         I_2MIN = MAX(⌈(II_2 + FirstIter^ZU(pid, I_1)) * β − φ⌉+2, 2)
         I_2MAX = MIN(⌈(II_2 + FirstIter^ZU(pid, I_1) + 1) * β − φ⌉+2, N)-1
         DO I_2 = I_2MIN TO I_2MAX
R^ZU:       ZU[I_2,I_1] = ZU[I_2,I_1] + ···
         END DO
      END DO
   END DO

   /* Loop nest containing ZV after splitting main loop */
   β = β^ZV(k)
   DO II''_2 = 0 TO ⌈(N − 2)/β⌉-1 by P
      DO I_1 = 2 TO 6
         φ = φ^ZV_I1(n)

         /* All elements of ZV written during iterations (I_1, I_2MIN : I_2MAX)
            lie within some k-block that is mapped to processor pid */
         I_2MIN = MAX(⌈(II_2 + FirstIter^ZV(pid, I_1)) * β − φ⌉+2, 2)
         I_2MAX = MIN(⌈(II_2 + FirstIter^ZV(pid, I_1) + 1) * β − φ⌉+2, N)-1
         DO I_2 = I_2MIN TO I_2MAX
R^ZV:       ZV[I_2,I_1] = ZV[I_2,I_1] + ···
         END DO
      END DO
   END DO

END DOALL
```

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 P_j, where $0 \leq P_j < P$, which provides some affinity across DOALL loops.

The compiler generated three versions of each Fortran 77 benchmark studied: ORIG, PP-OPT and FS-OPT. For the ORIG and PP-OPT versions, each processor was assigned a consecutive chunk of $\beta = N/P$ iterations, where N is the problem size. For the PP-OPT version, a buffer size of β was chosen. 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 all three versions, the innermost DOALL loop was parallelized. To maximize the grain of parallelism, loop interchanging was then applied when legal.

5.1 DMXPY

Loop Nest 11 depicts the Fortran kernel DMXPY from LINPACKD [7] which performs matrix–vector multiplication.

Loop Nest 11 /* 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
```

Fig. 4 depicts the performance of the ORIG, PP-OPT and FS-OPT versions of these programs for four different problem sizes. As can be seen in Fig. 4(a), the overhead for applying either PP-OPT or FS-OPT is less than 10% of the sequential execution time. Therefore, as the number of processors increases and the degree of false sharing with it, the optimized versions quickly outperform ORIG. Note that the curves that correspond to the optimized versions are smoother as well. This makes the performance of the optimized versions easier to predict, which facilitates program tuning.

For this benchmark and the range of processors studied, the performance of PP-OPT and FS-OPT is similar. The only exception occurs when the number of processors is very small in comparison to the problem sizes. In this case, the degree of false sharing is too small to offset the load unbalancing caused by FS-OPT. However, this trend quickly reverses as the number of processors is increased. This effect can be seen in Fig. 4(c)(d).

5.2 Triangularized DMXPY

Because there is processor affinity across executions of the $\mathtt{I_1}$ loop in DMXPY, the reference pattern is the same on every execution of this loop, and the degree

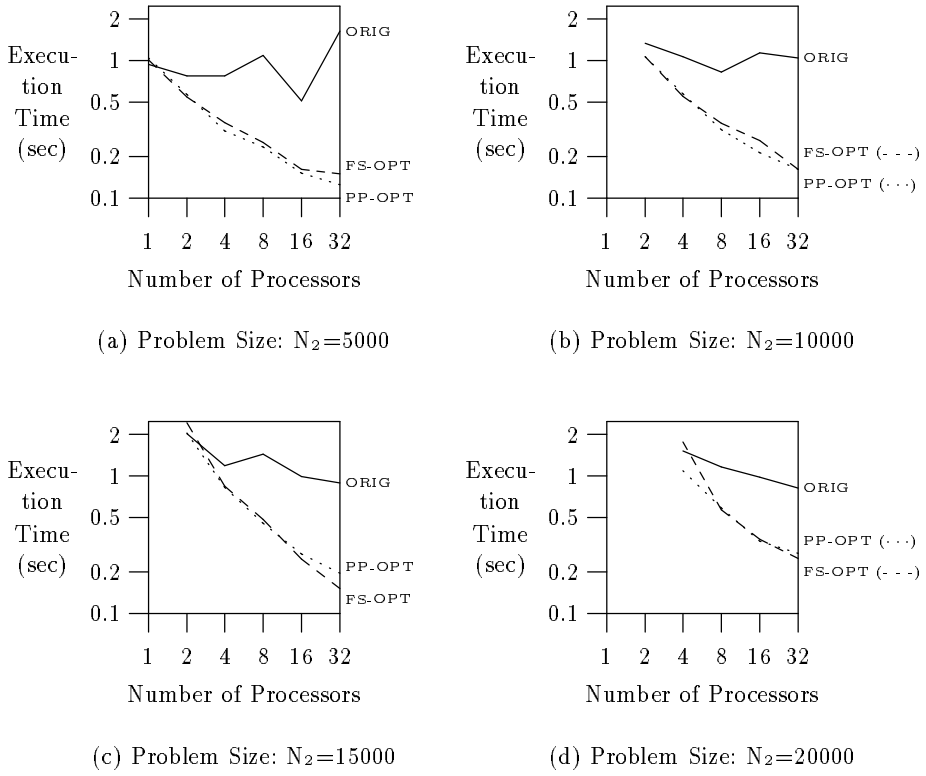

Fig. 4. *Execution times for original and optimized versions of* DMXPY *(Loop Nest 11) with the inner loop parallelized and interchanged.* $\mathtt{N_1} = 10$.

of false sharing is not very high. False sharing would become more significant if the reference pattern changed across executions of the $\mathtt{I_1}$ loop. To create such a situation, we created the triangularized version of the DMXPY loop shown in Loop Nest 12.

Loop Nest 12 *(* Triangularized version of* DMXPY *)*

```
      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 the triangularized version of DMXPY can be seen in Fig. 5. Again, the optimized versions outperform the unoptimized versions. This time, however, the FS-OPT version significantly outperforms the PP-OPT version. There are several reasons for this. First, because of the triangulation, the PP-OPT version no longer has the advantage of affinity across DOALL loop

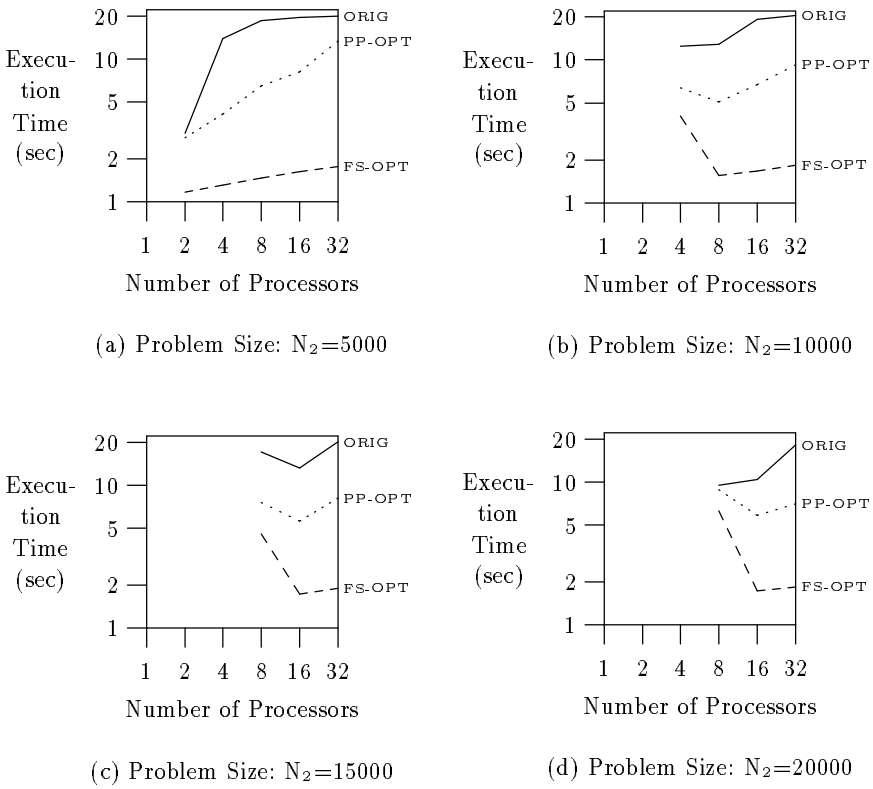

Fig. 5. *Execution times for unoptimized and optimized triangularized versions of* DMXPY *(Loop Nest 12) with inner loop parallelized (no interchanging).* $\mathtt{N_1} = 100$.

executions, but the FS-OPT version still does. Second, the number of iterations per processor and, hence, the buffer size decrease as $\mathtt{I_1}$ increases. Because the benefits of PP-OPT are proportional to the buffer size, PP-OPT becomes less effective as execution of the triangularized kernel progresses.

5.3 LLK18

Fig. 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 contains three loop nests similar to that depicted in Loop Nest 7. 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 eliminated with respect to one write reference in each loop nest, it is automatically eliminated with respect to both.

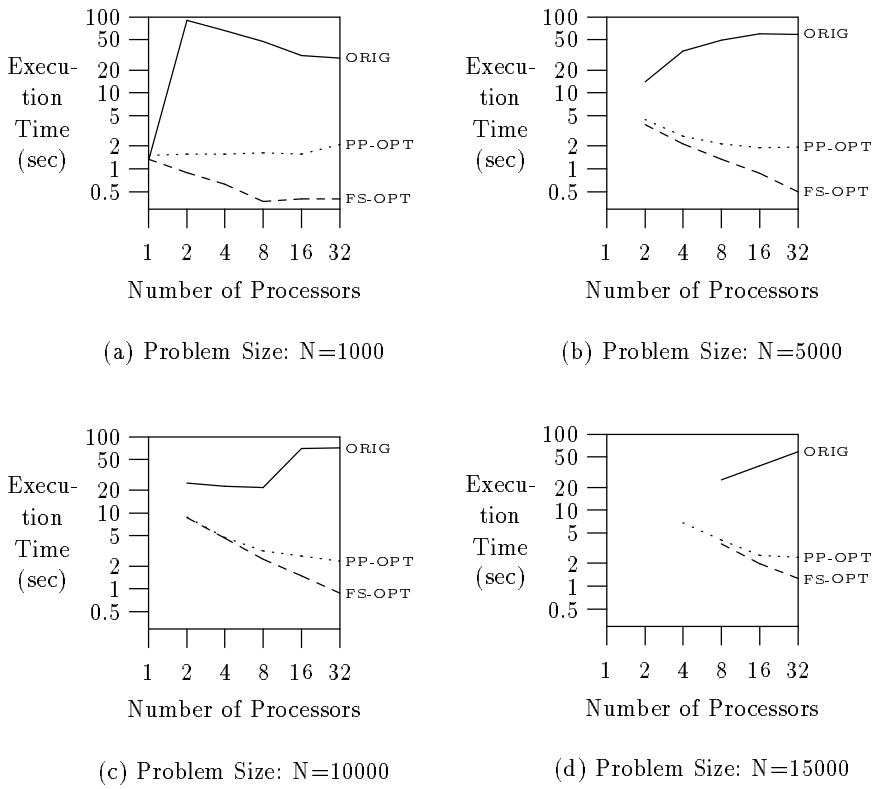

Fig. 6. *Execution times for original and optimized versions of* LLK18*, with inner loops parallelized and interchanged. All arrays are aligned with page boundaries, so there is only one reference group per loop nest.*

As can be seen in Fig. 6, both PP-OPT and FS-OPT again significantly outperform ORIG. As the number of processors increases, FS-OPT increasingly outperforms PP-OPT. Note that this is true for all four graphs in Fig. 5 as well. 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 as indivisible units (i.e., all writes to a given page must be performed 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.

In contrast, PP-OPT does not make any such requirement. Therefore, as the number of processors increases, the number of iterations per processor decreases. Because the maximum effective buffer size is bounded from above by the number of iterations per processor, the maximum effective buffer size decreases as well. Because the benefits of PP-OPT are proportional to the buffer size, PP-OPT

becomes less effective as the number of processors is increased. Note that, for these experiments, we set the actual buffer size equal to the maximum possible buffer size. Regardless of the buffer size chosen, however, the same trend would be observed beyond some threshold.

The best example of this effect can be seen in Fig. 6(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 Fig. 6 as well.

5.4 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 7 so that they no longer match `ZU`. In Fig. 7, we compare the original version of this loop nest (ORIG) to the three optimized versions PP-OPT, FS-PP-OPT, and FS-OPT, depicted as Loop Nests 8, 9 and 10. For ORIG, PP-OPT and FS-OPT, block and buffer sizes are chosen as described earlier. For FS-PP-OPT, we use the same block size as in FS-OPT, and the buffer size is chosen to be equal to the block size.

In general, for 2 to 32 processors, all three optimized versions greatly outperform ORIG. Note that, in the case of both PP-OPT and FS-PP-OPT, the maximum possible buffer size shrinks as the number of processors increase. In PP-OPT, the shrinkage is effectively unbounded for the reasons mentioned earlier. Therefore, the performance benefits of PP-OPT will eventually diminish as the number of processors is increased (this can already be seen for the two smaller problems sizes as the number of processors approaches 32). In both the FS-OPT and FS-PP-OPT versions, the amount of parallelism is bounded from above, because of the indivisibility of pages. Consequently, in FS-PP-OPT, the buffer size cannot grow smaller than $\beta(1)$, the smallest possible block size. Therefore, the performance curves for both these versions will more or less flatten out for the reasons mentioned earlier. FS-OPT slightly outperforms FS-PP-OPT, but the differences for this benchmark are negligible.

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 Fig. 7(c), ORIG performs very poorly on two processors because of thrashing. Because of the smaller working sets of the optimized versions, all three perform much better.

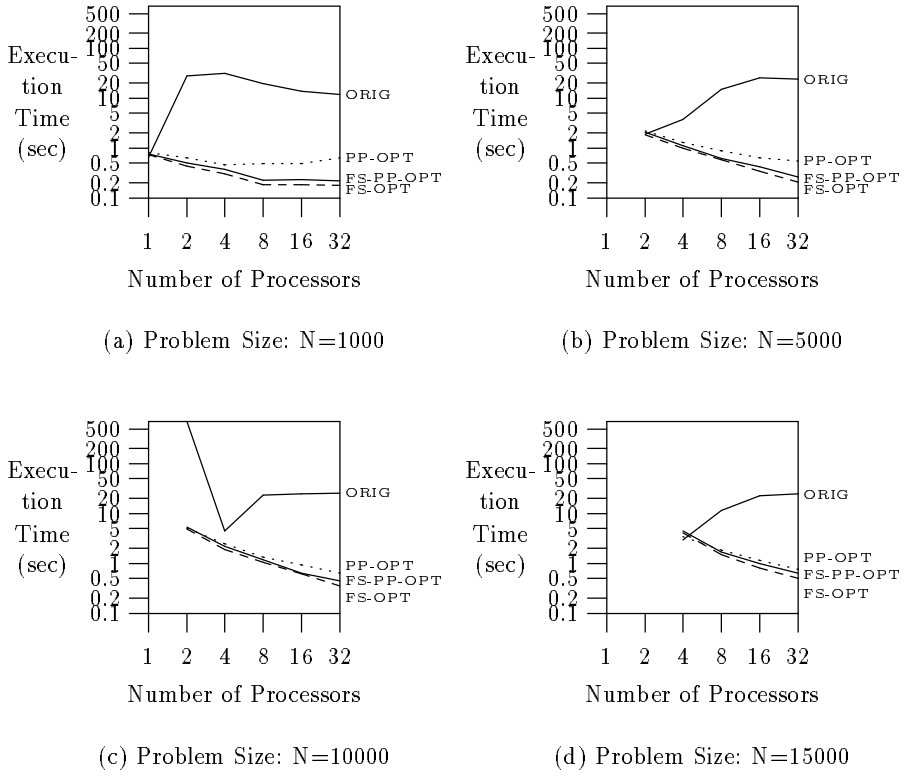

Fig. 7. *Execution times for original and optimized versions of Loop Nest 7 from* LLK18*, with inner loops parallelized and interchanged.* `ZU` *and* `ZV` *are declared with different dimensions, so that Loop Nest 7 contains two reference groups.*

6 Related Research

The potential performance degradation that can be caused by false sharing has been studied by several researchers. Based on this research, data layout optimizations to reduce false sharing, such as array padding [2, 8, 9] and array reindexing [10], have been proposed. Others have studied data layout optimizations to reduce false sharing in languages with structures and pointers [11].

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 [12, 13, 14] as well as optimizations that increase locality across loop nests, for example [15, 16, 17]. Unfortunately, when the coherency unit becomes larger, such techniques no longer suffice.

One run-time solution to the false sharing problem is to relax the consistency model. Systems such as Treadmarks [18] (by default) and KOAN [2] (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.

Observe that, 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. Therefore, we do 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 eliminated to ensure correctness. Breternitz et al. [19] study this problem. Tthey 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 eliminating false sharing.

Eliminating false sharing is only one optimization that a good SVM compiler should perform. Several researchers have begun exploring techniques to reduce synchronization overhead and hide latencies on SVM systems [10, 16, 20].

7 Conclusions

In this paper, we have distinguished between false sharing and its chief symptom, the ping-pong effect. We have evaluated two compile-time techniques for attacking multiple-writer false sharing for the purpose of improving performance (as opposed to ensuring program correctness). The first is PP-OPT, a new loop transformation that reduces the ping-pong effect by encouraging processors to perform multiple writes to a page before relinquishing the page. It is simple to implement and has wide applicability. We have shown that using this technique to reduce ping-pong effects can eliminate much of the performance degradation attributed to false sharing. For cases where a more aggressive approach is merited, we evaluated a second technique FS-OPT that targets false sharing directly. This loop transformation partitions iterations into blocks of "pages" and assigns these blocks to processors as indivisible units, thus ensuring that no page is written by more than one processor.

For both techniques, 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.

Overall, both techniques increase locality and, hence, reduce thrashing. 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. A secondary benefit to both optimizations is that performance becomes more predictable, which facilitates both manual and automatic program tuning.

In general, for any fixed problem size, increasing the amount of parallelism beyond some threshold will cause an application's performance to deteriorate. One of the benefits of FS-OPT is that, as a side effect of treating pages as indivisible units, the amount of parallelism that will be utilized is bounded from above by the number of pages. This helps prevent performance from deteriorating when the number of processors available becomes too large. In contrast, PP-OPT does not impose any limits on parallelism or even provide any hints as to what the bound should be.

Consequently, in general, when the number of processors is increased beyond this threshold, FS-OPT should increasingly outperform PP-OPT. Below this threshold, however, the performance of the two optimizations is expected to be comparable in many cases. These trends can be seen in the preliminary experimental data presented in this paper. Because this threshold is often difficult to determine, the profitability margin of FS-OPT is likely to be higher. Moreover, FS-OPT has the advantage of naturally providing affinity across doall loops. However, PP-OPT has the advantage over FS-OPT of being simpler to implement and more generally applicable. Therefore, an aggressive compiler should probably support both FS-OPT and PP-OPT; FS-OPT should be applied where possible and PP-OPT used for the remaining cases. In a less aggressive compiler, it would probably suffice to support PP-OPT only.

Because the performance degradation due to page migrations is proportional to page size, so is the benefit of applying our techniques. Although we targeted systems with page-sized coherency units, it should also be possible to realize (albeit smaller) performance gains on SVM systems with cache-line-sized coherency units.

Clearly, eliminating false sharing is only one of several optimizations that a good SVM compiler should perform. In the future, we hope to study and incorporate optimizations to reduce synchronization, to hide latencies where communication is unavoidable, and to handle irregular accesses.

Acknowledgements

The authors would like to thank Thierry Priol and Zakaria Lahjomri for providing access to and assistance with KOAN, Preston Briggs, Ervan Darnell, William Jalby, Ken Kennedy, Ajay Sethi and the anonymous referees for their helpful comments and suggestions, and Debbie Campbell for proofreading this paper.

François Bodin and Thierry Montaut are 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. Elana D. Granston is supported by the Center for Research on Parallel Computation under Grant No. CCR-9120008, a grant from the International Business Machines Corporation, and a Postdoctoral Research Associateship in Computational Science and Engineering under National Science Foundation Grant No. CDA-9310307.

References

1. K. Li, *Shared Virtual Memory on Loosely Coupled Multiprocessors.* PhD thesis,

Yale University, Sept. 1986.

2. Z. Lajormi and T. Priol, "KOAN: A Shared-Memory for the iPSC/2 Hypercube," in *CONPAR/VAPP92*, LNCS 634, Springer-Verlag, Sept. 1992.
3. T. Montaut and F. Bodin, "False Sharing in Shared Virtual Memory: Analysis and Optimization," tech. rep., IRISA, 1993.
4. F. Bodin, E. D. Granston, and T. Montaut, "Experiences Reducing False Sharing in Shared Virtual Memory Systems." Submitted for publication.
5. E. D. Granston, "Toward a Compile-Time Methodology for Reducing False Sharing and Communication Traffic in Shared Virtual Memory Systems," in *the* Sixth Annual Workshop on Languages and Compilers for Parallel Computing, (Portland, Oregon), Aug. 1993. Published in *Languages and Compilers for Parallel Computing*, Banerjee et al. (Eds.), LNCS 768, Springer–Verlag, 1994, pages 273–289.
6. F. Bodin, L. Kervella, and T. Priol, "Fortran-S: A Fortran Interface for Shared Virtual Memory Architectures," in *Supercomputing '93*, pp. 274–283, IEEE Computer Society Press, Nov. 1993.
7. J. Dongarra, J. Bunch, C. Moler, and G. Stewart, *LINPACK User's Guide*, 1979.
8. W. J. Bolosky, R. P. Fitzgerald, and M. L. Scott, "Simple But Effective Techniques for NUMA Memory Management," in *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pp. 19–31, ACM Press, Dec. 1989.
9. J. Torrellas, M. S. Lam, and J. L. Hennessy, "False Sharing and Spatial Locality in Multiprocessor Caches," Aug. 1992. Submitted to *IEEE Transactions on Computers*.
10. S. P. Ammarsinghe, J. M. Anderson, M. S. Lam, and C.-W. Tseng, "Design and Evaluation of Compiler Optimizations for Scalable Address Space Machines," 1994. To be published.
11. S. J. Eggers and T. E. Jeremiassen, "Eliminating false sharing," in *Proceedings of the International Conference on Parallel Processing*, pp. 377–381, CRC Press, Inc., Aug. 1991.
12. F. Bodin, C. Eisenbeis, W. Jalby, and D. Windheiser, "A Quantitative Algorithm for Data Locality Optimization," in *Code Generation-Concepts, Tools, Techniques*, Springer-Verlag, 1992.
13. K. Kennedy and K. S. McKinley, "Optimizing for Parallelism and Data Locality," in *International Conference on Supercomputing*, pp. 323–334, ACM Press, July 1992.
14. M. E. Wolf and M. S. Lam, "A Data Locality Optimizing Algorithm," in *Proceedings of the SIGPLAN '91 Conference on Programming Languages Design and Implementation*, pp. 30–44, ACM Press, June 1991.
15. J. Fang and M. Lu, "A Solution to the Cache Ping-Pong Problem in RISC Based Parallel Processing Systems," in *International Conference on Parallel Processing*, Aug. 1991.
16. B. Appelbe, C. Hardnett, and S. Doddapaneni, "Program Transformation for Locality Using Affinity Regions," in *the* Sixth Annual Workshop on Languages and Compilers for Parallel Computing, (Portland, Oregon), Aug. 1993. Published in *Languages and Compilers for Parallel Computing*, Banerjee et al. (Eds.), LNCS 768, Springer–Verlag, 1994, pages 290–300.
17. J. Anderson and M. 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.
18. 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.
19. M. Breternitz, Jr., M. Lai, V. Sarkar, and B. Simons, "Compiler Solutions for the Stale-Data and False-Sharing Problems," Tech. Rep. 03.466, IBM Santa Teresa Laboratory, Apr. 1993.
20. R. Michandaney, S. Hiranandani, and A. Sethi, "Improving the Performance of DSM Systems via Compiler Involvement," in *Supercomputing '94*, 1994.
21. L. Censier and P. Feautrier, "A New Solution to Coherence Problems in Multicache Systems," *IEEE Transactions on Computers*, pp. 1112–1118, Dec. 1978.

A Overview of KOAN

The KOAN SVM system [2] 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 [1], with an invalidation protocol that ensures that the shared memory is coherent at all times [21]. 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.

This article was processed using the LaTeX macro package with LLNCS style