Applying Array Contraction to A Sequence of DOALL Loops

Yonghong Song
Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054
yonghong.song@sun.com

Zhiyuan Li
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907
li@cs.purdue.edu

Abstract

Efficient program execution on multiprocessor computers requires both sufficient parallelism and good data locality. Recent research found that, using a combination of loop shifting, loop fusion, and array contraction, one can reduce the memory required to execute a sequence of serial loops, thereby to improve the cache locality. This paper studies how to extend such a memory-reduction scheme to a sequence of DOALL loops which are executed in parallel on multiprocessors. Two methods are proposed to overcome difficulties caused by loop-carried dependences. Data copy-in *is performed to remove anti-dependences between different parallel threads, and* computation duplication *is performed to remove flow dependences. Experiments performed on a number of benchmark programs show that the proposed technique improves both cache locality and parallel execution speed for the DOALL loops. The scheme achieves an average speedup of 1.41 for 17 programs on a 4-processor SUN machine.*

1 Introduction

Efficient program execution on multiprocessor computers requires both sufficient parallelism and good data locality. Recent research introduced a scheme (called *SFC*) that combines loop shifting (*S*), loop fusion (*F*) and array contraction (*C*) to transform a sequence of loop nests into a single loop nest such that the required total memory space is minimized [15, 14]. This reduction in memory requirement improves both the cache locality and the sequential execution speed of the program. It is then natural to ask whether the SFC can be used to improve the cache locality of a sequence of DOALL loops. (A DOALL loop is an indexed loop, e.g. a Fortran DO loop, which has no loop-carried data dependences, i.e., data dependences between different iterations, except those in a parallelizable reduction operation such as a vector summation.)

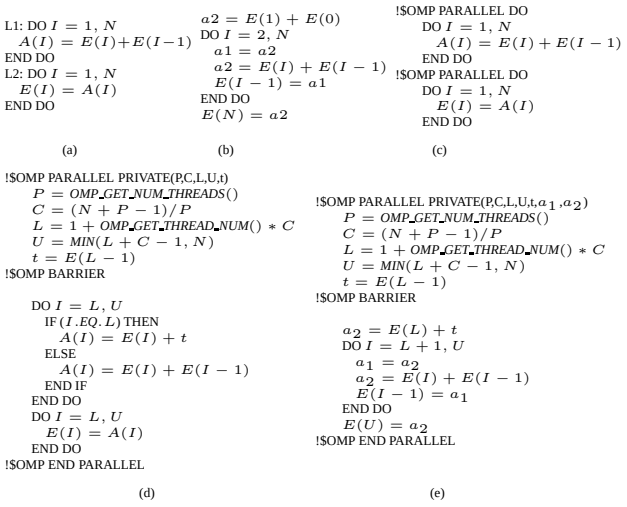


Figure 1. Example 1

In this paper, we shall show experimental results which demonstrate improved cache locality and execution speed of a number of benchmark programs as a result of applying a parallel version of SFC. Some of these programs require only straightforward loop fusion and array contraction. But for others, straightforward loop fusion (even with loop shifting) introduces loop-carried data dependences, which makes the resulting loop nest no longer DOALL. Hence, such programs require additional transforms in order to preserve the parallelism while allowing array contraction.

Figure 1 shows an example. Figure 1(a) lists a sequence of two loops, L1 and L2. Suppose array A is dead after loop L2. Using the SFC technique [15, 14], one can first shift the loop limits in L2, then fuse both loops before contracting the array A into two scalars (Figure 1(b)). On the other hand, to execute the loops in Figure 1(a) on a multiprocessor computer, one can convert L1 and L2 into parallel loops, since they are both DOALL. Figure 1(c) shows how to use OpenMP directives to mark these two loops as paral-

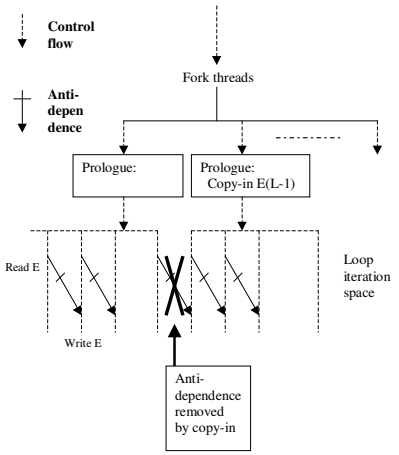


Figure 2. Illustration of Data Copy-In

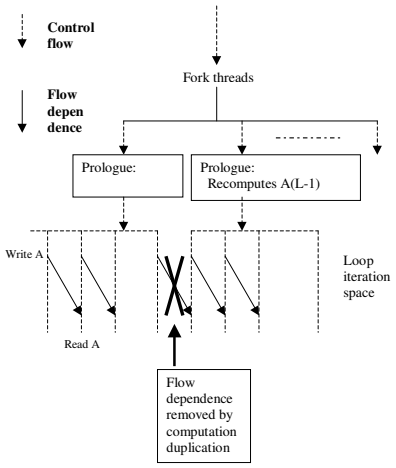


Figure 3. Illustration of Computation Duplication

lel. Unfortunately, the SFC technique cannot be applied in a straightforward way to these parallel loops, because they cannot be legally fused with or without loop shifting. (The fused loop in Figure 1(b) is not DOALL, since the shifting of L2 has created a loop-carried anti-dependence due to array E.) What we need is an alternative way to parallelize the given sequence of DOALL loops such that the SFC technique can be applied.

Our solution takes advantage of the fact that the number of available processors, P, is usually much smaller than the iteration counts of the parallel loops. We partition the iterations into P blocks of consecutive iterations. By letting each processor executes one of such blocks, we confine the potential data races (due to the loop-carried data dependences) to the array elements that are accessed in those iterations on the partition boundary. We can then remove such boundary dependences by two kinds of duplication. The anti-dependences are removed by *data copy-in*, which duplicates the array elements in contention. The flow-dependences are removed by *computation duplication*, which makes the reader of an array element recompute the data value instead of taking the value computed by a different processor. Note that, based on the assumptions made in Section 2, output dependences will not exist between different threads.

Figure 1(d) shows the parallelization of the loops in Figure 1(a) using the idea of data copy-in. The parallel code, which uses OpenMP directives, is executed by each processor which participates in the execution of the loops. Depending on its assigned thread number, the processor gets to execute a certain block of the iterations. The variable P represents the number of threads which, for the purpose of this paper, can be viewed as the number of available processors. The lower bound L and upper bound U define the range of the iterations executed in each thread. $E(L-1)$ is copied in to remove the anti-dependence between the boundary iterations. Such copy-in operations are placed in the parallel loop's prologue, which is separated from the thread body by a synchronization barrier. Figure 2 illustrates how the copy-in removes the boundary anti-dependences. The SFC technique [15, 14] can now be applied to the thread body, producing the final array-contracted code in Figure 1(e).

To see how *computation duplication* works, consider an example which is produced by replacing the reference to $A(I)$ in loop L2 (see Figure 1(a)) by a reference to $A(I-1)$. In this example, if the parallel code were generated following the style of Figure 1(d), there would exist a flow dependence between different processors. Figure 3 illustrates how to use computation duplication to remove such a flow dependence.

In the rest of the paper, we shall present a scheme which a compiler may use to perform array contraction on DOALL loops as illustrated above. We present experimental results obtained by applying the scheme to the benchmark programs used in the previous study of the SFC technique [15, 14]. In Section 2, we present a program model for this study. In Section 3, we present the new compiler scheme for a sequence of DOALL loops. We show experimental results in Section 4, compare related work in Section 5 and conclude in Section 6.

2 Program Model and Loop Dependence Graph

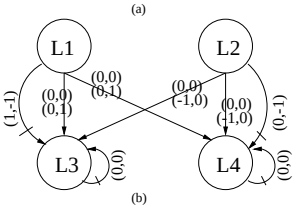
We consider a collection of loop nests, L_1, L_2, ..., L_m, $m \geq 1$, as shown in Figure 4. Each label L_i denotes a tight nest of loops with indices $L_{i,1}$, $L_{i,2}$, ..., $L_{i,n}$, $n \geq 1$, listed from the outermost level to the innermost. (For the exam-

L_1 : DO $L_{1,1} = l_{11}, l_{11} + b_1 - 1$
  DO $L_{1,2} = l_{12}, l_{12} + b_2 - 1$
  . . .
  DO $L_{1,n} = l_{1n}, l_{1n} + b_n - 1$
. . .
L_i : DO $L_{i,1} = l_{21}, l_{21} + b_1 - 1$
  DO $L_{i,2} = l_{22}, l_{22} + b_2 - 1$
  . . .
  DO $L_{i,n} = l_{2n}, l_{2n} + b_n - 1$
. . .
L_m : DO $L_{m,1} = l_{m1}, l_{m1} + b_1 - 1$
  DO $L_{m,2} = l_{m2}, l_{m2} + b_2 - 1$
  . . .
  DO $L_{m,n} = l_{mn}, l_{mn} + b_n - 1$

Figure 4. The program model

```
L1: DO K = 2, KN
      DO J = 2, JN
        ZA(J, K) = ZP(J − 1, K + 1)
          +ZR(J − 1, K − 1)
      END DO
      END DO
L2: DO K = 2, KN
      DO J = 2, JN
        ZB(J, K) = ZQ(J −1, K)+ZZ(J, K)
      END DO
      END DO
L3: DO K = 2, KN
      DO J = 2, JN
        ZP(J, K) = ZP(J, K) + ZA(J, K)
          −ZA(J − 1, K) − ZB(J, K)
          +ZB(J, K + 1)
      END DO
      END DO
L4: DO K = 2, KN
      DO J = 2, JN
        ZQ(J, K) = ZQ(J, K) + ZA(J, K)
          +ZA(J − 1, K) + ZB(J, K)
          +ZB(J, K + 1)
      END DO
      END DO
```

(a)



(b)

```
!$OMP PARALLEL PRIVATE(P,C,L,U)
P = OMP_GET_NUM_THREADS()
C = (KN − 1 + P − 1)/P
L = 2 + OMP_GET_THREAD_NUM() ∗ C
U = MIN(L + C − 1, KN)

DO K = L, U
  DO J = 2, JN
    ZA(J, K) = ZP(J − 1, K + 1)
      +ZR(J − 1, K − 1)
  END DO
END DO
DO K = L, U
  DO J = 2, JN
    ZB(J, K) = ZQ(J − 1, K) + ZZ(J, K)
  END DO
END DO
DO K = L, U
  DO J = 2, JN
    ZP(J, K) = ZP(J, K) + ZA(J, K)
      −ZA(J − 1, K) − ZB(J, K)
      +ZB(J, K + 1)
  END DO
END DO
DO K = L, U
  DO J = 2, JN
    ZQ(J, K) = ZQ(J, K) + ZA(J, K)
      +ZA(J − 1, K) + ZB(J, K)
      +ZB(J, K + 1)
  END DO
END DO
!$OMP END PARALLEL
```

(c)

Figure 5. Example 2, its original loop dependence graph and intermediate result before computation duplication and data copy-in

ple in Figure 1(a), we have $m = 2$ and $n = 1$.) All the outermost loops are assumed to be DOALL, with the possible presence of parallelizable reduction operations as mentioned before. We make a number of assumptions which are the same as in the previous study of the SFC technique for sequential execution [15, 14]. These assumptions are restated below.

Loop $L_{i,j}$ has the lower bound l_{ij} and the upper bound $l_{ij} + b_j - 1$ respectively, where l_{ij} and b_j are loop invariants. All loops at the same level, j, are assumed to have the same trip count b_j. We assume that none of the given loops can be partitioned into smaller loops by *loop distribution* [18]. Otherwise, we apply maximum loop distribution [18] to the given collection of loops first. The purpose of maximum loop distribution is for optimal memory-space reduction [15].

The array regions referenced in the given collection of loops are divided into three categories. An *input array region* is upwardly exposed to the beginning of L_1. An *output array region* is live after L_m. A *local array region* does not intersect with any input or output array regions. Only the local array regions are amenable to array contraction. In the example in Figure 1(a), $A(1 : N)$ is the only local array region. Figure 5(a) shows a more complex example which resembles one of the well-known Livermore loops. In this example, where $m = 4$ and $n = 2$, each declared array is of dimension $[1 : JN + 1, 1 : KN + 1]$. *ZA(2:JN,2:KN)* and *ZB(2:JN,2:KN)* are local array regions.

To describe the dependence between a collection of loop nests, we extend the definitions of the traditional dependence distance vector [6] as follows.

Definition 1 *Given a collection of loop nests, L_1, ..., L_m, as in Figure 4(a), if a data dependence exists from the source iteration $\vec{i} = (i_1, i_2, \ldots, i_n)$ of loop $L_{k_1}(1 \leq k_1 \leq m)$ to the destination iteration $\vec{j} = (j_1, j_2, \ldots, j_n)$ of loop $L_{k_2}(1 \leq k_2 \leq m)$, we say the* distance vector *of this dependence is $\vec{j} - \vec{i} = (j_1 - i_1, j_2 - i_2, \ldots, j_n - i_n)$.*

For simplicity of discussion, we assume that there exist no output dependences between different loop nests. Furthermore, we assume constant dependence distance vectors in this paper. In certain cases, one can replace non-constant distance vectors by constant ones without suffering the optimality of the solution [14].

Figure 5(b) illustrates the data dependences in the code example in Figure 5(a). The array regions associated with the dependence edges can be inferred from the program, and they are omitted in the figure. For instance, the flow dependence from L_1 to L_3 with $\vec{d} = (0, 0)$ is due to array region *ZA*$(2 : JN, 2 : KN)$. In Figure 5(b), where multiple dependences of the same type (flow, anti- or output) exist from one node to another, all these dependences are represented by a single arc. All associated distance vectors are then marked on this single arc.

3 The New SFC for DOALL Loops

The SFC technique relies on the fact that, given a sequence of loops shown in Figure 4(a), the number of simultaneously live array elements in the local array regions can be reduced by fusing the given m loop nests into a single loop nest. Through Figure 1, we have illustrated how to make loop fusion legal by loop shifting and how to contract arrays after loop fusion. The details of these steps and the optimal loop-shifting choice are discussed in previous studies [15, 14]. In this paper, we focus on the new aspects of SFC when applied to DOALL loops.

In our new scheme, the compiler first creates a parallel region around the given sequence of loop nests. The iterations of each loop nest are divided evenly among the

```
procedure find_comp_dup()
  /* Let R_i be the input array region and R_d the array region which needs to be recomputed. */
  R_d = φ.
  for i = m to 1 by −1 do
    for (each flow dependence whose destination is in L_i) do
      Let L_p (1 ≤ p < i) be the loop containing the dependence source.
      Determine the array region, R_w, written by the dependence source.
      Determine the array region, R_r, used by the dependence destination.
      R_d = R_d ∪ (R_r − R_w − R_i). /* Only the needed values that are computed
            by a different processor must be recomputed locally by duplication. */
    end for
  end for
end procedure
```

Figure 6. The algorithm for determining duplicated computation

```
procedure find_copy_in()
  /* Let R_o be the read-only input array region and R_d the array region which needs to be copied in. */
  R_d = φ.
  for i = 1 to m do
    for (each anti-dependence whose source is in L_i) do
      Determine the dependence destination L_j, j > i.
      Determine the array region, R_r, read by the dependence source in loop L_i.
      Determine the array region, R_w, written by the dependence destination in loop L_j.
      R_d = R_d ∪ (R_r − R_w − R_o).
    end for
  end for
end procedure
```

Figure 7. The algorithm for computing copy-in array regions

processors. Figure 5(c) shows the code for Figure 5(a) after this intermediate step. A synchronization barrier separates the prologue of the parallel region from the thread body, and the latter contains the original loop nests whose loop bounds are modified according to the partitioning. We then apply computation duplication and data copy-in to the loop nests in the thread body, if they are needed. Array contraction follows afterwards, regardless whether computation duplication and data copy-in are applied or not. Next, we discuss how to determine whether computation duplication and data copy-in are needed in order to perform array contraction and, if needed, how to apply them.

3.1 Computation Duplication

Computation duplication may have a ripple effect that must be handled systematically. A duplicated computation in loop L_i may require the values computed in another loop, say L_j, by a different processor. A new case of boundary flow dependence hence may arise. Note that loop L_j must precede L_i lexically in the given sequence of loops, hence $j < i$. A flow dependence from L_i to L_i will not require computation duplication because it cannot be loop-carried in $L_{i,1}$ which is a DOALL loop. Figure 6 shows the algorithm that formalizes the handling of the ripple effect. This algorithm visits the loop nests in the reversed lexical order. It determines all array regions whose computation must be duplicated because of the boundary flow dependences.

The inserted computation-duplication code recomputes the values of certain array elements that are used in the thread. The storage for such recomputed values deserves a careful consideration. In order to reduce the required memory size, it is beneficial to store these values in the original arrays. However, there are cases in which we are forced to allocate new variables that are private to the thread in order to store the recomputed values. Suppose an array element is written more than once in one of the original loop nests. It then has different values, at different time, that are used by different read references. Further suppose that, after loop parallelization, the same array element needs to be recomputed (by computation duplication) in the prologue of the parallel region in order to remove flow dependences between different threads. Without allocating private variables, the single storage for the original array element obviously cannot hold multiple values, even though all of these values are used by some read references in the same loop executed by the same thread. Private variables must therefore be allocated to store all these values. The read operations will then read the corresponding private variables.

For the example in Figure 5(c), we can easily find that the array region which needs recomputation is $ZB(2 : JN, U + 1)$. This is due to the flow dependence from the reference to $ZB(J, K)$ in L2 to the reference to $ZB(J, K + 1)$ in L3 and due to the flow dependence from the reference to $ZB(J, K)$ in L2 to the reference to $ZB(J, K + 1)$ in L4.

3.2 Data Copy-in

After using computation duplication to remove boundary flow dependences, we use data copy-in to remove boundary anti-dependences. It is possible for an array to be both written (for computation duplication) and read in the prologue, which creates data races. When this happens, we place all data copy-in code ahead of the computation duplication code in the prologue, and we use an additional synchronization barrier to separate these two parts.

Figure 7 shows the procedure for finding array regions which need to be copied in, where R_o represents the portion of input array regions which are read but not written in the given code segment. For each anti-dependence, the array region to be copied will be copied to variables that are private to the thread. Therefore such an anti-dependence no longer exists between different threads when the thread body is executed.

For the example in Figure 5(c), the array region which needs copy-in is $ZP(2 : JN, U + 1)$, which is due to the anti-dependence from the reference to $ZP(J - 1, K + 1)$ in L1 to the reference to $ZP(J, K)$ in L3.

3.3 Contracting the Arrays

After all necessary computation duplication and data copy-in are inserted in the parallel code region. The SFC

```
!$OMP PARALLEL PRIVATE(P,C,L,U,t)
P = OMP_GET_NUM_THREADS()
C = (KN − 1 + P − 1)/P
L = 2 + OMP_GET_THREAD_NUM() ∗ C
U = MIN(L + C − 1, KN)
t(1 : JN − 1) = ZP(1 : JN − 1, U + 1).
IF (U.NE.KN) THEN
  DO J = 2, JN
    ZB(J, U + 1) = ZQ(J − 1, U + 1)
      +ZZ(J, U + 1)
  END DO
END IF
!$OMP BARRIER

DO K = L, U
  DO J = 2, JN
    IF (K.EQ.U) THEN
      ZA(J, K) = t(J − 1)
        +ZR(J − 1, K − 1)
    ELSE
      ZA(J, K) = ZP(J − 1, K + 1)
        +ZR(J − 1, K − 1)
    END IF
  END DO
END DO
DO K = L, U
  DO J = 2, JN
    ZB(J, K) = ZQ(J − 1, K) + ZZ(J, K)
  END DO
END DO
DO K = L, U
  DO J = 2, JN
    ZP(J, K) = ZP(J, K) + ZA(J, K)
      −ZA(J − 1, K) − ZB(J, K)
      +ZB(J, K + 1)
  END DO
END DO
DO K = L, U
  DO J = 2, JN
    ZQ(J, K) = ZQ(J, K) + ZA(J, K)
      +ZA(J − 1, K) + ZB(J, K)
      +ZB(J, K + 1)
  END DO
END DO
!$OMP END PARALLEL
```

(a)

```
!$OMP PARALLEL PRIVATE(P,C,L,U,t,ZB1,a1,a2,b)
P = OMP_GET_NUM_THREADS()
C = (KN − 1 + P − 1)/P
L = 2 + OMP_GET_THREAD_NUM() ∗ C
U = MIN(L + C − 1, KN)
t(1 : JN − 1) = ZP(1 : JN − 1, U + 1).
IF (U.NE.KN) THEN
  DO J = 2, JN
    ZB(J, U + 1) = ZQ(J − 1, U + 1)
      +ZZ(J, U + 1)
  END DO
END IF
!$OMP BARRIER

DO J = 2, JN
  ZB1(J) = ZQ(J − 1, L) + ZZ(J, L)
END DO

DO K = L, U − 1
  a1 = ZA(1, K)
  DO J = 2, JN
    a2 = ZP(J − 1, K + 1)
      +ZR(J − 1, K − 1)
    b = ZQ(J − 1, K + 1) + ZZ(J, K + 1)
    ZP(J, K) = ZP(J, K) + a2 − a1
      −ZB1(J) + b
    ZQ(J, K) = ZQ(J, K) + a2 + a1
      +ZB1(J) + b
    a1 = a2
    ZB1(J) = b
  END DO
END DO

a1 = ZA(1, K)
DO J = 2, JN
  a2 = t(J − 1) + ZR(J − 1, K − 1)
  ZP(J, K) = ZP(J, K) + a2 − a1
    −ZB1(J) + ZB(J, U + 1)
  ZQ(J, K) = ZQ(J, K) + a2 + a1
    +ZB1(J) + ZB(J, U + 1)
  a1 = a2
END DO
!$OMP END PARALLEL
```

(b)

Figure 8. Parallelized code for Example 2

technique [15, 14] is applied to the new loop nests L_1 though L_m in the thread body.

For the example in Figure 5(a), Figure 8(a) shows the code after automatic parallelization by applying computation duplication and data copy-in techniques. The only write reference to *shared* data is $ZB(J, U+1)$ in the computation duplication code and the data copy-in code. Since there exists no other read references to array *ZB* in those codes, there exists no data race between different threads.

Figure 8(b) shows the code after array contraction. For arrays *ZA* and *ZB* in Figure 8(a), the local array regions accessed are $ZA(1 : JN, L : U)$ and $ZB(2 : JN, L : U+1)$ respectively. After array contraction (Figure 8(b)), *ZA* is contracted to $ZA(1, L : U)$ plus two scalars, a_1 and a_2. Array *ZB* is contracted to $ZB(2 : JN, U+1)$, $ZB1(2 : JN)$ and a scalar b.

We do not apply the extended SFC, including computation duplication and data copy-in, unless we determine that the predicted performance after final array contraction is better than that of the original code segment. For simplicity, we assume that the synchronization overhead is the same before and after the transformation. (Although we introduce new barriers in the prologue of the parallel region, we may also remove certain barriers because of loop fusion.) With such an assumption, we predict the profitability of the extended SFC by examining whether it reduces the number of cache misses [15]. To estimate cache misses, we estimate the reuse distance [19] for each dependence and compare it to the cache size. If it is equal to or smaller than the cache size, the memory access at the dependence destination is assumed to be a cache hit. Otherwise, it is assumed to be a cache miss. We illustrate using Example 2 as follows.

Let C_s represent the cache size and C_b the cache line size, both measured in the number of data elements. We estimate the total number of cache misses in the original code of Example 2 (Figure 5(a)) by $x_1 = (KN-1) * (JN-1) * 12/C_b$. The total number of cache misses in the copy-in and computation duplication codes (accumulated over all threads) is estimated by $x_2 = (JN-1) * 5 * P/C_b$, where P represents the number of OpenMP threads. The number of cache misses in the code generated after array contraction (accumulated over all threads) is estimated by $(JN-1) * 2 + (U-L+1) * (JN-1) * 4 * P/C_b$, which is equivalent to $x_3 = ((JN-1) * 2 * P + (KN-1) * (JN-1) * 4)/C_b$. The condition $x_2 + x_3 < x_1$ holds if and only the condition $7 * P < (KN-1) * 8$ holds, which is the case for large *KN*. We can let the compiler generate two versions of code such that only under the condition $7 * P < (KN-1) * 8$ does the parallelized version with array contraction get executed.

4 Experimental Results

To evaluate the effectiveness of the array contraction technique discussed in this paper, we examine their applicability to the test programs used in the previous study of the SFC technique [15]. Among those 20 test programs, we find that the technique in this paper is not suitable for `LL14`, `lucas` and `laplace-gs`. This is because these three programs contain loops which are not DOALL but whose fusion is required in order to perform array contraction. For the experimentation in this paper, therefore, we exclude these three programs.

Table 1 lists the remaining 17 programs used in our experiments. In the listed programs, the loop nests fused for array contraction are all DOALL loops. In this table, "m/n" represents the number of loops in the loop sequence (m) and the maximum loop nesting level (n). For each of the benchmarks in Table 1, all m loops are fused together. For `swim95`, `swim00` and `hydro2d`, where $n = 2$, only the outer loops are fused. For all other benchmarks, all n loop levels are fused. Further details concerning the program descriptions and input parameter selections can be found in the previous SFC study [15].

Among the 17 programs, the DOALL loops in `combustion`, `climate`, and all those purdue-set programs can be directly fused into a single DOALL loop without inserting data copy-in and computation duplication, as shown in Table 1. Optimal array contraction can be applied

Table 1. Test Programs

Benchmark Name	Description	Input Parameters	m/n	Comp. duplication and copy-in
LL18	Livermore Loop No. 18	N = 400, ITMAX = 100	3/2	Computation Duplication
Jacobi	Jacobi Kernel w/o convergence test	N = 1100, ITMAX = 1050	2/2	Both
tomcatv	A mesh generation program from SPEC95fp	reference input	5/1	Both
swim95	A weather prediction program from SPEC95fp	reference input	2/2	Computation Duplication
swim00	A weather prediction program from SPEC2000fp	reference input	2/2	Computation Duplication
hydro2d	An astrophysical program from SPEC95fp	reference input	10/2	Both
mg	A multigrid solver from NPB2.3-serial benchmark	Class 'W'	2/1	Neither
combustion	A thermochemical program from UMD Chaos group	N1 = 10, N2 = 10	1/2	Neither
purdue-02	Purdue set problem02	reference input	2/1	Neither
purdue-03	Purdue set problem03	reference input	3/2	Neither
purdue-04	Purdue set problem04	reference input	3/2	Neither
purdue-07	Purdue set problem07	reference input	1/2	Neither
purdue-08	Purdue set problem08	reference input	1/2	Neither
purdue-12	Purdue set problem12	reference input	4/2	Neither
purdue-13	Purdue set problem13	reference input	2/1	Neither
climate	A two-layer shallow water climate model from Rice	reference input	2/4	Neither
laplace-jb	Jacobi method of Laplace from Rice	ICYCLE = 500	4/2	Both

without hurting the parallelism. In program `mgrid`, the sequence of DOALL loops fused for array contraction are all immediately embedded in an outermost DOALL loop. The remaining programs, i.e. `LL18`, `Jacobi`, `tomcatv`, `swim95`, `swim00`, `hydro2d` and `laplace-jb`, require computation duplication and/or data copy-in in order to keep the loops parallelized while allowing array contraction.

We manually apply the technique described in Section 3 to parallelize the loops using data copy-in and computation duplication. We then apply the tool developed in previous work [15] to perform loop shifting, loop fusion and array contraction (i.e. SFC). For comparison, we also run the parallelized code with computation duplication and data copy-in inserted, and then with loops fused. These versions, however, do not have arrays contracted. Altogether, thus, we have five versions of code to compare, including the original sequential code and the sequential code produced by the SFC technique. All these codes are run on a 4-processor SUN multiprocessor computer. Each processor is a SUN UltraSPARC II 248MHz processor with a 16KB L1 data cache and a 1MB L2 cache. The L1 cache is directly-mapped with a line size of 16 bytes. The L2 cache is directly-mapped with a line size of 64 bytes.

To generate the machine code, we use SUN's native Fortran compiler which has a version of Sun Workshop 6 update 1. All sequential codes are compiled with the "-fast" option. All parallel (OpenMP) codes are compiled with the "-fast -openmp -autopar" option. To measure the parallel performance of the original codes, we take the better execution time from two versions of codes. In one version, we manually add OpenMP pragmas to the original codes, which are compiled with the "-fast -openmp -autopar" option. The other version is parallelized automatically by the native compiler using its "-fast -autopar" option. The automatic parallelization facility of the compiler may re-arrange

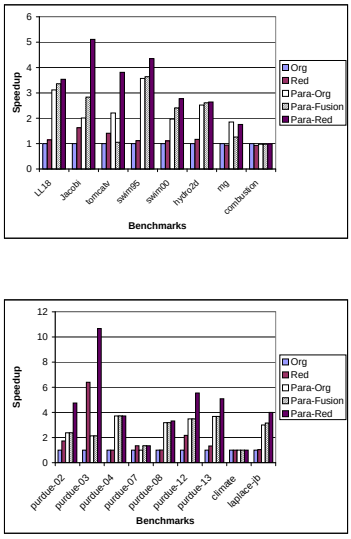


Figure 9. Execution results

loop nests for better performance. We add "-autopar" to the compilation flag such that the loops not fused by our technique can be examined by the native compiler for automatic parallelization. For programs `purdue-07` and `climate`, we turn off parallelization for both the original loop nests and the loop nests transformed by our technique. This is because of small trip counts and small amount of work in the loop body.

4.1 Execution Results

Figure 9 compares the execution speed. The label "Org" stands for the sequential execution of the original codes, which is used as the normalization base. The label "Red" stands for the serial execution of the codes transformed by SFC. The label "Para-Org" stands for the parallel execution of the original codes. "Para-Fusion" stands for the parallel execution of the code transformed by the new technique

except that the arrays are not yet contracted after loop fusion. "Para-Red" stands for the parallel execution of the final codes transformed by the new technique.

According to Figure 9, the parallelized codes transformed by our technique perform better than the parallelized original codes in all 17 programs except `mg`, with a speedup ranging from 0.95 to 5.0 and a geometric means of 1.41. For `mg`, the original codes perform better than memory-reduction codes in both single-processor and multiprocessor runs. We suspect that this abnormal behavior is caused by the instruction scheduling performed by the native compiler's back-end. We will investigate this behavior further in the future study.

According to Figure 9, for all those listed programs, the parallelized code with array contraction also performs better than the parallelized code after loop fusion but before array contraction. The speedup ranges from 1.01 to 5.0 with a geometric mean of 1.42. For `tomcatv`, the performance of the parallelized version after loop fusion but before array contraction is particularly poor. This is because, for this program, loop interchange is performed to enable loop fusion. Before array contraction, the loop interchange spoils the spatial locality of most of the arrays.

Figure 10 shows the L1 and L2 cache miss rates. We measure the cache miss rate by using the **perfmon** package. We only measure the cache miss rate for the main thread, since we have not found a reliable way to measure the cache miss rates for the entire parallel programs on the computer system used in the experiments. In this figure, "Para-Org-L1" and "Para-Org-L2" stand for the L1 and the L2 cache miss rates of the original codes parallelized in the straightforward fashion. "Para-Red-L1" and "Para-Red-L2" stand for the L1 and the L2 cache miss rates of the parallel codes with array contraction. From this figure, we see that for most of the programs, array contraction indeed reduces cache miss rate for either the L1 or the L2 cache, or both. For `swim00` and `climate`, although the miss rates are not reduced, the number of array references are reduced significantly because of several arrays being contracted to scalars. For `swim00`, the number is reduced from 9.2B to 6.3B. For `climate`, it is reduced from 6.9M to 1.5M.

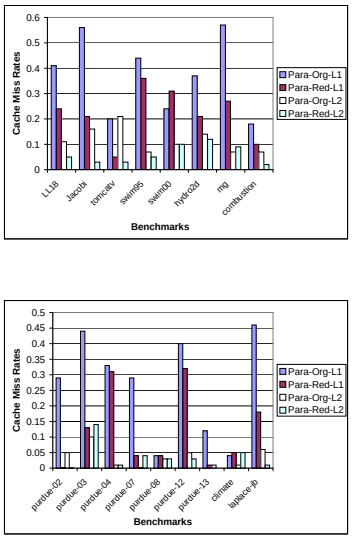


Figure 10. Cache miss rate

5 Related Work

Several authors have considered a single perfect nest of loops. For such a single loop nest, they have studied the relationship between loop schedules (including parallel schedules) and storage optimization [17, 16, 11]. Unlike the SFC technique and its parallel extension discussed in this paper, the previous studies consider neither loop fusion nor loop shifting to enable array contraction. Among these studies, Strout *et al.* first propose the use of a universal occupancy vector (UOV) to derive schedule-independent storage mapping for loops [16]. When such a mapping is found, arrays may be contracted along a single dimension without imposing new constraints on permissible loop schedules. Thie *et al.* study several problems concerning schedule and storage optimization [17], including the problems of finding the minimum storage under a given schedule, finding an optimal schedule (if possible) given an amount of storage, and finding the minimum storage that do not impose any new constraints on loop schedules. Pike *et al.* give a detailed study of the relationship between loop tiling and array contraction, given a perfect loop nest that is executed sequentially [11].

The computation duplication to avoid synchronization is not new. Mellor-Crummey *et al.* have used a similar computation duplication technique in dHPF compiler for effectively parallelization of HPF codes [9]. They target a single loop nest while we target a collection of loop nests. For cross-iteration anti-dependences, we use data copy-in while they use *selective* loop distribution to minimize synchronization overhead.

The SFC technique is first used to improve the performance of sequentially executed loops [15]. A closely related work is presented for the purpose of reducing the storage requirement in embedded systems [3]. Both of these techniques utilize network flow algorithms to compute the optimal shifting to minimize the required memory space, but the work in [3] targets a single-level loop only. The SFC technique [15] handles a collection of multi-level loop nests.

For loop fusion and array contraction, Kennedy and McKinley prove that maximizing data locality by loop fusion for registers is NP-hard [5]. Ding and Kennedy prove that loop fusion for maximum cache reuse is Np-hard [2]. Singhai and McKinley present *parameterized loop fusion* to improve parallelism and cache locality simultaneously [13].

Manjikian and Abdelrahman present a *shift-and-peel* technique to increase opportunities for loop fusion [8], which is first presented as *peel-and-jam* by Porterfield [12]. Allen *et al.* first combine loop distribution and loop fusion for parallelization purpose [1]. Gao *et al.* combine loop fusion and array scalarization to improve register utilization [4]. Lim *et al.* combine loop blocking, loop fusion and array contraction to exploit parallelism [7]. The above two works do not use loop shifting. Ng *et al.* combine loop fusion, array contraction and array rotation in a production compiler [10], which mostly focuses on single processor performance.

6 Conclusion

In this paper, we have presented a technique to apply array contraction to a sequence of DOALL loops. Previous studies have examined the relationship between parallel loop schedules and storage optimization for a single perfectly-nested loop. Our work presents a new method to combine array contraction with loop parallelization when given a sequence of DOALL loops. For loops that cannot be fused in a straightforward way to allow array contraction, we present two techniques, i.e. data copy-in and computation duplication, to remove fusion-preventing data dependences. Experimental results show that our technique obtain performance that is superior to the straightforward method which parallelizes the original loop nests without fusion and array contraction.

Acknowledgment

This work is sponsored in part by National Science Foundation through grants ITR/ACR-0082834 and CCR-0208760. The authors thank the reviewers for their careful reviews and useful suggestions.

References

[1] Allen, Callahan, and Kennedy. Automatic decomposition of scientific programs for parallel execution. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 63–76, January 1987.

[2] C. Ding and K. Kennedy. Improving effective bandwidth through compiler enhancement of global cache reuse. *Journal of Parallel and Distributed Computing*, 64(1):108–134, January 2004.

[3] A. Fraboulet, G. Huard, and A. Mignotte. Loop alignment for memory accesses optimization. In *Proceedings of the Twelfth International Symposium on System Synthesis*, Boca Raton, Florida, November 1999.

[4] G. R. Gao, R. Olsen, V. Sarkar, and R. Thekkath. Collective loop fusion for array contraction. In *Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing*. Also in No. 757 in *Lecture Notes in Computer Science*, pages 281–295, Springer-Verlag, 1992.

[5] K. Kennedy and K. S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *Springer-Verlag Lecture Notes in Computer Science, 768. Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland, Oregon, August, 1993.

[6] D. J. Kuck. *The Structure of Computers and Computations*, volume 1. John Wiley & Sons, 1978.

[7] A. W. Lim, S.-W. Liao, and M. S. Lam. Blocking and array contraction across arbitrarily nested loops using affine partitioning. In *Proceedings of 2001 ACM Conference on PPOPP*, pages 103–112, Snowbird, Utah, June 2001.

[8] N. Manjikian and T. Abdelrahman. Fusion of loops for parallelism and locality. *IEEE Transactions on Parallel and Distributed Systems*, 8(2):193–209, February 1997.

[9] J. Mellor-Crummey, V. Adve, B. Broom, D. Chavarria-Miranda, R. Fowler, G. Jin, K. Kennedy, and Q. Yi. Advanced optimization strategies in the rice dhpf compiler. *Concurrency - Practice and Experience*, 1:1–20, 2001.

[10] J. Ng, D. Kulkarni, W. Li, R. Cox, and S. Bobholz. Inter-procedural loop fusion, array contraction and rotation. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, pages 114–124, New Orleans, Louisiana, September 2003.

[11] G. Pike and P. N. Hilfinger. Better tiling and array contraction for compiling scientific programs. In *Proceedings of the IEEE/ACM SC 2002 Conference*.

[12] A. Porterfield. *Software Methods for Improving Cache Performance*. PhD thesis, Department of Computer Sciences, Rice University, May 1989.

[13] S. K. Singhai and K. S. McKinley. A parameterized loop fusion algorithm for improving parallelism and cache locality. *The Computer Journal*, 40(6), 1997.

[14] Y. Song, R. Xu, C. Wang, and Z. Li. Improving data locality by array contraction. *IEEE Transactions on Computers*. To appear in vol 53 no. 8, August 2004.

[15] Y. Song, R. Xu, C. Wang, and Z. Li. Data locality enhancement by memory reduction. In *Proceedings of the 15th ACM International Conference on Supercomputing*, Naples, Italy, June 2001.

[16] M. M. Strout, L. Carter, J. Ferrante, and B. Simon. Schedule-independent storage mapping for loops. In *Architectural Support for Programming Languages and Operating Systems*, pages 24–33, 1998.

[17] W. Thies, F. Vivien, J. Sheldon, and S. P. Amarasinghe. A unified framework for schedule and storage optimization. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 232–242, 2001.

[18] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Publishing Company, 1995.

[19] Y. Zhong, C. Ding, and K. Kennedy. Reuse distance analysis for scientific programs. In *Proceedings of the Sixth Workshop on Languages, Compilers, and Run-time Systems for Scalar Computers*, March 2002.