Improving Data Locality by Array Contraction

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

Rong Xu Cheng Wang Zhiyuan Li
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907
{xur,wangc,li}@cs.purdue.edu

Abstract

Array contraction is a program transformation which reduces array size while preserving the correct output. In this paper, we present an aggressive array-contraction technique and study its impact on memory system performance. This technique, called *controlled SFC*, combines *loop shifting* and *controlled loop fusion* to maximize opportunities for array contraction within a given loop nesting. A controlled fusion scheme is used to prevent over-fusing loops and to avoid excessive pressure on the cache and the registers. Reducing the array size increases data reuse because of the increased average number of memory operations on the same memory addresses. Furthermore, if the data size of a loop nest fits in the cache after array contraction, then repeated references to the same variable in the loop nest will generate cache hits, assuming set conflicts are eliminated successfully.

Index terms: compiler, memory, optimization, performance
Key words: array contraction, data locality, loop shifting, optimizing compilers

1 Introduction

Array contraction is a program transformation which reduces array size while preserving the correct output. This technique has been used in the past to reduce memory requirement of the program [11], which can be important to out-of-core computing and embedded systems. A special case of array contraction called *array scalarization* has also been known to improve register utilization [3, 12]. In this paper, we use array contraction to improve data reuse and cache locality. We transform an array to one or several lower-dimensional arrays with a

smaller total size. We present a technique which combines *loop shifting* (S), *loop fusion* (F) and array contraction (C) to minimize the memory required by arrays used in a given loop nesting. We call such a combination *SFC* in this paper. We use loop shifting to eliminate fusion-preventing data dependences, thereby creating more opportunities for loop fusion and array contraction.

Furthermore, we study the effect of array contraction on data reuse and cache locality. We use a scheme to control loop fusion such that we do not overly fuse loops and cause excessive pressure on the cache and the registers. Array contraction is applied to the resultant loop nests after this controlled fusion. We call the overall technique *controlled SFC*.

The rest of the paper is organized as follows. In Section 2, we discuss previous work. In Section 3, we formulate the mathematical framework for memory requirement minimization via the the SFC technique. We present the controlled SFC scheme and summarize our experimental results in Section 4. We conclude in Section 5. Mathematical proofs and details of experimental results are included in the Appendices.

2 Previous Work

Loop fusion has been studied extensively for improving parallelism, data locality, or both. Kennedy and McKinley prove that maximizing data locality by loop fusion is NP-hard [19]. Kennedy presents a heuristic method with complexity $O(V(E+V))$ [18]. Singhai and McKinley present *parameterized loop fusion* to improve parallelism and cache locality simultaneously [31]. Darte uses loop shifting to enable maximum fusion of parallel loops which have constant dependence distances [6]. His goal is to find the minimum number of partitions such that the loops within each partition can be shifted and fused and the fused

loop remains parallel. Manjikian and Abdelrahman present a *shift-and-peel* technique to increase opportunities for loop fusion [24]. Tembe and Pande use loop distribution and loop fusion to reduce off-chip data I/O traffic on embedded processors [34]. Since these previous publications do not contract arrays, their problem formulation is different from ours.

Gao *et al.* combine loop fusion and array scalarization to improve register utilization [12]. Lim *et al.* combine loop blocking, loop fusion and array scalarization to exploit parallelism [22]. They do not consider either partial contraction or loop shifting. Thies *et al.* analyze the tradeoff between parallelism and storage allocation [35]. They study how to optimize storage allocation without jeopardizing parallel task scheduling. Lefebvre and Feautrier present a technique to remove all anti- and output dependences by fully expanding the arrays such that parallelism is maximized [21]. They then try to re-contract the arrays without sacrificing parallelism. With the exception of the work by Lim *et al.*, these previous efforts do not perform loop fusion. Furthermore, since their objectives are different from ours, the problem formulation is also different.

Several researchers have proposed combining loop and data-layout transformations to improve spatial locality without losing temporal locality [7, 17, 23]. These techniques do not reduce array size.

Fraboulet *et al.* present an algorithm to minimize the array size required to execute a collection of single-level loops which can be immediately fused into a single loop [11]. They require the data dependence distances to be known constants. After loop fusion, their algorithm realigns the loop iterations for each statement in the loop body in order to minimize the total size of arrays. They reduce the problem of optimal realignment to a network-flow problem which can be solved in polynomial time. Their network-flow formulation does

not apply to multi-level loops. Neither does it apply to *coalesced* single-level loops. This is because the loop bounds will become parts of the data dependence distances and loop bounds usually are not known constants. They use a heuristic to handle multi-level loops by processing one loop level at a time. If there exist fusion-preventing data dependences, they fuse only those loops which are not involved in such dependences. We should note that the work by Fraboulet *et al.* does not discuss data locality on cache-based memory systems. Instead, their interest is on minimizing memory requirement in embedded-system design.

Our recent SFC technique [33] uses a network-flow algorithm to reduce the memory requirement for a collection of multi-level loop nests, each of which is perfectly nested. This technique has several advantages over that of Fraboulet et al. It uses loop shifting to remove fusion-preventing data dependences, thus creating new opportunities for loop fusion and array contraction. Moreover, under a few conditions, this polynomial-time technique ensures memory minimization not only for single-level loops, but also for multi-level loops.

In this paper, we improve our work in [33] by formulating the controlled SFC technique in a clearer and more accessible way. We present details of the main concepts and assumptions for the underlying mathematical framework. Moreover, since the relationship between array contraction and locality enhancement has not been closely examined in existing work, we study their relationship by a combination of analysis, simulation and measurement.

3 Minimizing Memory Requirement by SFC

Opportunities for array contraction exist often because the most natural way to specify a computational task may not be the most memory-efficient, and also because the programs written in array languages such as F90 and HPF are often memory inefficient. The SFC

technique looks for such opportunities by identifying array variables whose lifetime can be shortened through loop fusion. Take the Jacobi relaxation code in Figure 1(a) as an example. Within each T-iteration, the four-neighbor averages of array A are computed and stored in array *temp*. It requires $(N-2)^2$ elements of *temp* to store such temporary results in each T-iteration. We wish to fuse the loop nests labeled by L_1 and L_2 such that a smaller number of *temp* elements are required. In order to preserve the data dependences, however, loop shifting must be applied before fusion.

When applying loop shifting to a loop whose control variable is i, the appearance of i in the loop body is replaced by $i - p_i$, $p_i > 0$. Assuming unit stride, the lower and the upper bounds are both incremented by p_i. The number p_i is called the *shifting factor* for the i-loop. Figure 1(b) shows the code after shifting the J_2 loop by the shifting factor of 1. The loops can now be fused as shown in Figure 1(c). Notice that the outer loop, J, of the fused loop nest has expanded its index domain to $[2, N]$. Two IF-conditions, called *guards*, are inserted in the loop body. One guard makes sure that the statement from the original L_1 loop body does not get executed in iteration $J = N$. The other guard makes sure that the statement from the original L_2 loop body does not get executed in iteration $J = 2$. After loop fusion, the lifetime of each element of array *temp* is shortened, in the sense that between its write and read only $N - 2$ distinct *temp* elements remain live. The array *temp*, therefore, can be contracted to a new array, *temp*$'$, which has a single dimension and the size of $N - 2$ elements. In addition, a scalar r is needed to temporarily store the four-neighbor average before it can be safely copied to an element of *temp*$'$. Figure 1(d) shows the code after array contraction.

In the rest of this section, we develop a memory-minimization problem under loop shifting,

```
      DO T = 1, ITMAX
L1:   DO J1 = 2, N − 1
        DO I1 = 2, N − 1
          temp(I1, J1) = (A(I1 + 1, J1)
          +A(I1 − 1, J1) + A(I1, J1 + 1)
          +A(I1, J1 − 1))/4
        END DO
      END DO
L2:   DO J2 = 2, N − 1
        DO I2 = 2, N − 1
          A(I2, J2) = temp(I2, J2)
        END DO
      END DO
      IF (converge) exit
    END DO
```

(a)

```
      DO T = 1, ITMAX
L1:   DO J1 = 2, N − 1
        DO I1 = 2, N − 1
          temp(I1, J1) = (A(I1 + 1, J1)
          +A(I1 − 1, J1) + A(I1, J1 + 1)
          +A(I1, J1 − 1))/4
        END DO
      END DO
L2:   DO J2 = 3, N
        DO I2 = 2, N − 1
          A(I2, J2 − 1) = temp(I2, J2 − 1)
        END DO
      END DO
      IF (converge) exit
    END DO
```

(b)

```
DO T = 1, ITMAX
DO J = 2, N
  DO I = 2, N − 1
    IF (J.EQ.2) THEN
      temp(I, J) = (A(I + 1, J)
        +A(I − 1, J) + A(I, J + 1)
        +A(I, J − 1))/4
    ELSE IF (J.EQ.N) THEN
      A(I, J − 1) = temp(I, J − 1)
    ELSE
      temp(I, J) = (A(I + 1, J)
        +A(I − 1, J) + A(I, J + 1)
        +A(I, J − 1))/4
      A(I, J − 1) = temp(I, J − 1)
    END IF
  END DO
END DO
IF (converge) exit
END DO
```

(c)

```
DO T = 1, ITMAX
DO J = 2, N
  do I = 2, N − 1
    IF (J.EQ.2) THEN
      temp′(I) = (A(I + 1, J)
        +A(I − 1, J) + A(I, J + 1)
        +A(I, J − 1))/4
    ELSE IF (J.EQ.N) THEN
      A(I, J − 1) = temp′(I)
    ELSE
      r = (A(I + 1, J)
        +A(I − 1, J) + A(I, J + 1)
        +A(I, J − 1))/4
      A(I, J − 1) = temp′(I)
      temp′(I) = r
    END IF
  END DO
END DO
IF (converge) exit
END DO
```

(d)

Figure 1: The Jacobi example

loop fusion and array contraction. We first explain how the shifting factors determine the required temporary memory to execute a given set of loop nests. We then present the program model and state our assumptions. Based on these, we formalize the mathematical framework.

3.1 Preliminaries

A loop nest is said to be *perfectly nested* if all its executable statements, except the loop control statements, are embedded in its unique innermost loop. We represent an instance of the loop body by a vector $\vec{i} = (i_1, i_2, \ldots, i_n)$, where n is the depth of the loop nest. The

element i_k is the value of the k-th loop control variable for that particular instance. The indices are ordered from the outermost loop to the innermost.

Suppose w is a reference which writes to a distinct element of array A in each loop iteration. The number of distinct A elements written by w during the execution of loop iterations between $\vec{i}$ and $\vec{j}$, $\vec{j} \succ \vec{i}$, is equal to the inner product of $\vec{j} - \vec{i}$ and the *coalescing vector* defined below [1].

Definition 1 *For the perfectly-nested loops given above, suppose b_k is the trip count of the loop at level k. We define the* coalescing vector *of the given loop nest to be $\vec{s} = (s_1, s_2, \ldots, s_n)$, such that $s_n = 1$, $s_k = s_{k+1} b_{k+1}, 1 \leq k \leq n-1$.*

Clearly, s_k equals the number of times the innermost loop body is executed during each iteration of the loop at the k-th level. For any pair of iteration vectors $\vec{i}$ and $\vec{j}$ such that $\vec{j} \succeq \vec{i}$, we call the inner product $(\vec{j} - \vec{i})\vec{s}^T$ their *coalesced distance*. This distance equals the number of times the innermost loop body is executed between $\vec{i}$ and $\vec{j}$.

Suppose there exists a data dependence between w and a reference r such that the value written by w in iteration $\vec{i} = (i_1, i_2, \ldots, i_n)$ is used by r in iteration $\vec{j} = (j_1, j_2, \ldots, j_n)$. Such data dependence from a write to a read is also called *flow dependence*. By convention [36], this data dependence is said to have a distance vector of $\vec{j} - \vec{i} = (j_1 - i_1, j_2 - i_2, \ldots, j_n - i_n)$. For convenience, we say that the data dependence has a *coalesced distance* equal to $(\vec{j} - \vec{i})\vec{s}^T$.

We assume all the values written by w are useful, meaning that they will be accessed by some read references. Before the value written by w in iteration $\vec{i}$ is used by r in iteration $\vec{j}$,

[1] Following notations in [36], given $\vec{u} = (u_1, u_2, \ldots, u_n)$ and $\vec{v} = (v_1, v_2, \ldots, v_n)$, we write $\vec{u} + \vec{v} = (u_1 + v_1, u_2 + v_2, \ldots, u_n + v_n)$, $\vec{u} - \vec{v} = (u_1 - v_1, u_2 - v_2, \ldots, u_n - v_n)$, $\vec{u} \succ \vec{v}$ if $\exists 0 \leq k \leq n-1, (u_1, \ldots, u_k) = (v_1, \ldots, v_k) \wedge u_{k+1} > v_{k+1}$, $\vec{u} \succeq \vec{v}$ if $\vec{u} \succ \vec{v}$ or $\vec{u} = \vec{v}$, and lastly, $\vec{u} > \vec{v}$ if $u_k > v_k$ $(1 \leq k \leq n)$. In this paper, we also use $\vec{u}\vec{v}^T$ to represent the inner product of $\vec{u}$ and $\vec{v}$, and use $abs(\vec{u})$ to represent $(|u_1|, |u_2|, \ldots, |u_n|)$.

w writes to a number of other elements of array A. This number is the minimum array size required to satisfy the flow dependence between w in $\vec{i}$ and r in $\vec{j}$. For convenience, we call this number the *memory-space cost* of the flow dependence. (As in Jacobi, a single scalar is needed to temporarily store the value written by w in iteration $\vec{j}$. We do not count this scalar.) From the discussion above, the memory-space cost of the flow dependence between w in $\vec{i}$ and r in $\vec{j}$ is equal to the coalesced distance of this dependence. Suppose that w is the only write reference in the loop nest which writes to array A and that A is dead after the entire loop nest is executed. The minimum size of A required to execute this loop nest is then equal to the maximum of the memory-space cost among all the flow dependences which have w as the source.

Next, consider a sequence of two loop nests labeled L_1 and L_2 respectively, each being perfectly nested. We assume that both L_1 and L_2 have the same nesting depth, n, and the same trip count b_k at each loop level k. Suppose that there exists a flow dependence from a write reference w in L_1 and a read reference r in L_2 such that the value written by w in iteration $\vec{i}$ is used by r in iteration $\vec{j}$. If there exist no data dependences (including flow, anti-, and output dependences) between L_1 and L_2 to prevent loop fusion, then, as in the Jacobi example, we may reduce the memory-space cost of the flow dependence by fusing the two loop nests. Since $\vec{j} - \vec{i}$ will become the distance vector between these two iterations after loop fusion, we call this difference vector the *predicted distance vector*, or in short, the *distance vector*.

Notice that we do not require that the two loop nests before fusion have identical loop bounds at the same loop level, although we require the trip counts to be the same. Therefore, when we try to determine the memory-space cost after loop fusion, we must be careful with

the loop bounds at all loop levels. Although the fused loop has an expanded index domain, reference w mentioned above is not issued in all iterations of the fused loop, because of the inserted guards. If $\vec{j}$ mentioned above is within the bounds of the original loop nest L_1, then it is not difficult to verify that the memory-space cost of the flow dependence between w and r is equal to the coalesced distance of the dependence from $\vec{i}$ to $\vec{j}$. In contrast, if $\vec{j}$ falls out of bound of L_1, then the memory-space cost will be less than the coalesced distance.

Take Figure 2 for example. After loop fusion, the dependence distance between the write and the read references of A equals a constant vector $(1, 1)$. The coalesced distance of the dependence equals $N - 1$. The memory-space cost of the flow dependence between the write of $A(2, 2)$ and the read is equal to its coalesced distance. However, the memory-space cost of the flow dependence between the write of $A(N - 1, N - 1)$ and the read equals one only. The read of $A(N, N)$ occurs in iteration (N, N), which falls out of the bounds of the original loop nest that contains the write reference.

Suppose that w is the only write reference in the fused loop nest which writes to array A, that all values written by w are useful, and that A is dead after the entire fused loop nest is executed. The minimum required size of A is then equal to the maximum of the memory-space cost among all the flow dependences which have w as the source. In the example in Figure 2, the maximum memory-space cost equals $N - 1$.

Next, we discuss what to do if there exist data dependences from L_1 to L_2 which make loop fusion illegal. Since L_2 follows L_1 immediately, it is a well-known fact that both loop nests can be fused if and only if every data dependence, say from iteration $\vec{u}$ of L_1 to iteration $\vec{v}$ of L_2, satisfies $\vec{v} \succeq \vec{u}$ [36]. In other words, their dependence distance vector must be lexicographically nonnegative. If there exist any data dependence which does not

```
DO J1 = 2, N − 1
  DO I1 = 2, N − 1
    A(I1, J1) = ...
  END DO
END DO
DO J2 = 3, N
  DO I2 = 3, N
    ... = A(I2 − 1, J2 − 1)
  END DO
END DO
```

(a) Before fusion

```
DO J = 2, N
  DO I = 2, N
    IF ((I < N).AND.(J < N))
      A(I, J) = ...
    IF ((I > 2).AND.(J > 2))
    ... = A(I − 1, J − 1)
  END DO
END DO
```

(b) After fusion

Figure 2: Reuse Distances Vary for Different Instances of Data Dependence

satisfy this condition, then fusing L_1 and L_2 will cause the source and the destination of the dependence to be reversed, making the fusion illegal.

Suppose a fusion-preventing dependence exists from memory operation [2] r_1 in iteration $\vec{u}$ of L_1 to memory operation r_2 in iteration $\vec{v}$ of L_2. To make loop fusion legal, we need to perform loop shifting on L_2 using a proper vector of shifting factors $\vec{p} = (p_1, p_2, \ldots, p_n)$ such that p_k is applied to loop level k. The dependence destination in L_2 after loop shifting becomes $\vec{v'} = \vec{v} + \vec{p}$. The requirement on $\vec{p}$ is that $\vec{v'} \succeq \vec{u}$. Obviously, there exist multiple choices of $\vec{p}$. Among them, the optimal choice is the one which is lexicographically minimum. This is because under this choice, the memory-space cost of the flow dependence from any write w to its dependent read r will be minimum after loop fusion. Any other legal choice of the shifting vector can only increase the number of times the innermost loop body executes between w and r. In Figure 2(a) $(1, 1)$ is the lexicographical minimum among all fusion-enabling shifting vectors applied to L_2.

The key optimization problem in this paper is to find the lexicographical minimum among all fusion-enabling shifting vectors such that, after loop shifting and loop fusion, the minimum

[2] In this paper, we call a run-time instance of a variable reference a *memory operation*, assuming the variable is allocated to the memory instead of a register. A memory operation may be a write operation or a read operation.

memory requirement to execute the fused loop nest is minimized. For an arbitrary number of sets of perfectly-nested loops, such a problem can be formulated as a set of integer programming problems. However, the lexicographical minimum is difficult to work with when one tries to develop a polynomial-time algorithm. In the rest of this section, we explain under certain assumptions how we can use the coalescing vector to formulate our optimization problem as a single integer programming problem. Such an integer programming problem can be reduced to a network-flow problem for which polynomial algorithms exist. Due to space limitations, we refer readers to our work elsewhere [33] for the transformations which lead to the network-flow problem. Here, we focus on the formulation of the single integer programming problem.

3.2 Loop Coalescing

Our main idea is to *imagine* that we perform *loop coalescing* on the given set of perfectly-nested loops. Loop coalescing is a loop transformation performed by mapping an iteration vector $\vec{i} = (i_1, i_2, \ldots, i_n)$ to its *coalesced index* defined as the inner product of $\vec{i}$ and $\vec{s}$. The multi-level loop nest is transformed to a single-level loop whose iteration domain is the range of the coalesced index. The advantage of this approach is that a lexicographically negative dependence distance vector may become lexicographically nonnegative after loop coalescing, which makes it legal to fuse the coalesced loops. This nice property allows us to formulate the optimization problem as a set of linear integer constraints. Theorem 1 presented later in this section states that, under certain assumptions, the minimum memory-space cost is completely determined by coalesced distances. Thus, if two choices of shifting vectors result in the same coalesced distances for all data dependences, then they both minimize the

memory-space cost. According to the following lemma, we do not need to actually coalesce the loops.

Lemma 1 *For* any *set of shifting vectors, we can find a set of shifting vectors which result in the same coalesced distances for all data dependences and they allow* legal *fusion of the loop nests* before *coalescing.*

Proof See Appendix A. □

Figure 3(b) shows the result of applying loop shifting to Figure 3(a) using the shifting vector of $(0, N-1)$. Such a shifting vector is illegal for the purpose of loop fusion, since the loop nests in Figure 3(b) still cannot be fused. However, after coalescing the loop nests in Figure 3(b) to two single-level loops in Figure 3(c), they can be fused. It is not difficult to verify that the memory-space cost of flow dependences for array *temp*, which equals to $N-1$, is the minimum under any combination of loop shifting, loop coalescing and loop fusion. On the other hand, we can find that the fusion-enabling shifting vector $(1, 1)$ can be applied to Figure 3(a) to achieve the same memory-space cost after fusion.

Let $\vec{b} = (b_1, b_2, \ldots, b_n)$ be the trip counts. The following lemma establishes an important relationship between the iteration vector $\vec{i}$ and its coalesced index.

Lemma 2 $\forall \vec{u}$ *s.t.* $abs(\vec{u}) < \vec{b}$, $\vec{u} \succ \vec{0} \Leftrightarrow \vec{u}\vec{s}^T > 0$.

Proof See Appendix B. □

Suppose $\vec{i_1}$ and $\vec{i_2}$ are two different iterations in a perfectly-nested loop. Since $abs(\vec{i_2} - \vec{i_1}) < \vec{b}$ holds, Lemma 2 immediately derives the following equivalence:

$$\vec{i_2} \succ \vec{i_1} \Leftrightarrow \vec{i_2}\vec{s}^T > \vec{i_1}\vec{s}^T. \tag{1}$$

```
L1: DO J = 2, N - 1                    L1: DO J = 2, N - 1
      DO I = 2, N - 1                        DO I = 2, N - 1
        temp(I, J) = A(I - 1, J - 1)           temp(I, J) = A(I - 1, J - 1)
      END DO                                 END DO
    END DO                                 END DO
L2: DO J = 2, N - 1                    L2: DO J = 2, N - 1
      DO I = 2, N - 1                        DO I = N + 1, 2N - 2
        A(I, J) = temp(I, J)                   A(I - N + 1, J) = temp(I - N + 1, J)
      END DO                                 END DO
    END DO                                 END DO

        (a)                                    (b)

L1: DO J1 = 2(N - 2) + 2, (N - 1)(N - 2) + N - 1
      J = DIV(J1 - 2(N - 2) - 2, N - 2) + 2
      I = MOD(J1 - 2(N - 2) - 2, N - 2) + 2
        temp(I, J) = A(I - 1, J - 1)
    END DO
L2: DO J1 = 2(N - 2) + N + 1, (N - 1)(N - 2) + 2N - 2
      J = DIV(J1 - 2(N - 2) - N - 1, N - 2) + 2
      I = MOD(J1 - 2(N - 2) - N - 1, N - 2) + N + 1
        A(I - N + 1, J) = temp(I - N + 1, J)
    END DO

        (c)
```

Figure 3: An example of loop coalescing

The equivalence property (1) means that loop coalescing preserves the execution order of all loop iterations and that the mapping between $\vec{i}$ and its coalesced index is one-to-one. Furthermore, the coalesced index values form a consecutive sequence, because the difference between the upper limit and the lower limit equals $(\vec{b} - \vec{1})\vec{s}^T + 1 = b_1 b_2 \ldots b_n$, which is the total number of instances of the innermost loop body before coalescing. Given the coalesced index and the trip counts $\vec{b}$, we can recompute $\vec{i}$ using MOD and divide operations. (We omit the derivation details here.)

At this point, we should note that, given a single set of perfectly-nested loops, it may be possible to reduce its memory requirement by first distributing it into two or more sets of perfectly-nested loops. After that, we can find shifting vectors for those sets of loops such that they can be fused back into one set with minimized memory requirement. In the following, we assume that we have already performed maximum loop distribution such

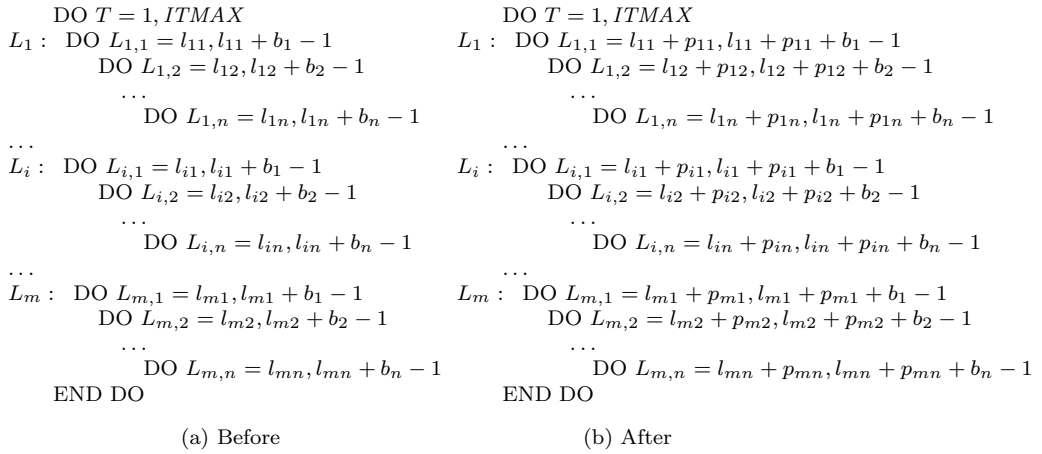


Figure 4: Loop nests before and after loop shifting

that each set of perfectly-nested loops can no longer be distributed. Compiler methods for maximum loop distribution are well-known [36].

3.3 Program Model

We consider an iterative loop T with a trip count $ITMAX$, which contains a collection of loop nests, L_1, L_2, ..., L_m, $m \geq 1$, in the lexical order shown in Figure 4(a). If loop T does not exist, the compiler may proceed as if there exists an artificial T loop with $ITMAX = 1$. The label L_i denotes a perfect nest of loops with indices $L_{i,1}$, $L_{i,2}$, ..., $L_{i,n}$, $n \geq 1$, starting from the outmost loop. We assume that all loops $L_{i,j}$ at level j have the same trip count b_j with the lower bound l_{ij} and the upper bound $l_{ij} + b_j - 1$ respectively, where l_{ij} and b_j are constants or symbolic constants within the loop nesting. Without loss of generality, we assume all loops $L_{i,j}$ have a step of 1. For simplicity of presentation, all the loop nests L_i, $1 \leq i \leq m$, are assumed to have the same nesting level n. If this is not the case, we can apply controlled SFC to the highest n_{min} levels, where n_{min} is the least nesting depth among all loop nests. The array regions referenced in the given collection of loops are divided into the following three classes. 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.

By utilizing the existing dependence analysis, region analysis and live analysis techniques [5, 8, 10, 13, 15], we can compute input, output and local array regions efficiently. Note that input and output regions may overlap with each other. In the Jacobi example (Figure 1), *temp*(2:N-1,2:N-1) is the local array region, A(1:N,1:N) is the input region, and A(2:N-1,2:N-1) is the output region.

Definition 2 *A* loop dependence graph *(LDG) is a directed multi-graph* $G = (V, E)$ *such that each node in* V *represents a loop nest* L_i, $1 \leq i \leq m$, *in Figure 4(a). (We write* $V = \{L_1, L_2, \ldots, L_m\}$*.) Each directed edge in* E *represents a data dependence (flow, anti- or output dependence) from one loop nest, say* L_i*, to another, say* L_j*. Each edge* e *is annotated by its* predicted distance vector $\vec{d_e}$*, or in short, its* distance vector *(Section 3.1).*

Given an iteration vector $\vec{i}$ in the iteration domain of L_i, its dependent iteration $\vec{j}$ in L_j can be expressed as an n-element function vector defined over $\vec{i}$. It is possible that certain elements of $\vec{j}$ remain undefined for a particular $\vec{i}$ because of the absence of dependence. Clearly, $d_e = \vec{j} - \vec{i}$ can also be expressed as an n-element function vector defined over $\vec{i}$. Existing techniques for data dependence analysis [10, 36] can be utilized to construct the LDG. This analysis takes exponential time in the worst case but it is quite fast in common cases. As discussed earlier, we use the dependence distance vectors to determine the legality of loop fusion and the memory-space cost. In our controlled SFC scheme, we first optimistically fuse all of the m loop nests in Figure 4(a) into a single loop nest and find all contractable arrays.

Applying our previous discussions to the entire collection of loop nests, we immediately derive the following legality condition.

Lemma 3 *It is legal to fuse all of the m loop nests in Figure 4(a) into a single loop nest if and only if*

$$\vec{d_e} \succeq \vec{0}, \forall e \in E. \tag{2}$$

We wish to find optimal loop shifting such that after loop shifting and loop coalescing, all loop nests can be fused into a single-level loop and the memory requirement is minimized. For this purpose, we associate each loop $L_{i,j}$ with a shifting factor p_{ij}. The shifting vector for the loop nest labeled L_i is then $\vec{p_i} = (p_{i1}, p_{i2}, \ldots, p_{in})$. The code after loop shifting is shown in Figure 4(b). For any dependence edge $e = \langle L_i, L_j \rangle$ with the distance vector $\vec{d_e} = \vec{j} - \vec{i}$ before loop shifting, the new dependence distance vector after loop shifting is $(\vec{j} + \vec{p_j}) - (\vec{i} + \vec{p_i}) = \vec{p_j} - \vec{p_i} + \vec{d_e}$. After applying loop coalescing to the loops in Figure 4(b), the new code will consist of m single-level loops. For any $\vec{d_e} = \vec{j} - \vec{i}$ before loop shifting, the *coalesced dependence distance* after loop coalescing is $(\vec{j}+\vec{p_j})\vec{s}^T - (\vec{i}+\vec{p_i})\vec{s}^T = (\vec{p_j} - \vec{p_i} + \vec{d_e})\vec{s}^T$. Applying Lemma 3 to the loops after coalescing, we immediately have the following lemma.

Lemma 4 *After applying loop shifting and loop coalescing described above to the loop nests in Figure 4(a), it is legal to fuse the loops into a single loop if and only if*

$$(\vec{p_j} + \vec{d_e} - \vec{p_i})\vec{s}^T \geq 0, \forall e = \langle L_i, L_j \rangle \in E. \tag{3}$$

The lower bound of the fused loop is the minimum of the lower bounds of the coalesced loops before fusion, and the upper bound is the maximum of the upper bounds. As discussed previously, guards are inserted in the fused loop to make the statements skip the iterations in which they must not be executed.

3.4 Formulating the Memory Minimization Problem

Before we formulate the integer programming problem to minimize memory requirement through array contraction, we make the following assumptions to simplify the discussion.

Assumption 1 *For each reference w which writes to a local array region, we assume that w writes to a distinct array element in every iteration of the innermost loop body. We assume that all values written by w are useful, meaning that each written value will be used by some read references in the given set of loop nests.*

If a reference w writes to the same memory location in different iterations, we apply array expansion [9, 36] to w as long as correct data flow can be preserved after expansion. Notice that the expanded array regions still remain local. If the assumptions in the rest of this subsection are also met, then our scheme will find all the contractable arrays and minimize the memory requirement. Obviously, the memory required will be no greater than before the array expansion.

Assumption 1 stated above implies that we exclude the cases in which w may not be executed in certain iterations. For such cases, we may need to take the IF conditions into account in order to determine the memory-space cost. We also exclude the cases in which a reference may write to local regions of an array in certain loop iterations and to non-local regions of the same array in other iterations. Array contraction can still be performed in such cases, but the memory-space cost must be calculated differently for the flow dependences. Moreover, the transformed code will be more complex.

Notice that if there exists an extremely large shifting factor, then the coalesced distance of a flow dependence may exceed $\Pi_{k=1}^{n} b_k$, which is the total iteration count of the innermost

loop body. Under Assumption 1, the memory-space cost of each instance of flow dependences is equal to either the coalesced distance after shifting or $\Pi_{k=1}^{n} b_k$, whichever is smaller. The following lemma determines the memory required to satisfy all flow dependences which have the same source w. This amount of memory is called the *memory-space cost* of w.

Lemma 5 *For the reference w in L_i which writes to a local array region, under Assumption 1, the memory required to satisfy all flow dependences which have w as the source equals*

$$\min(\max\{(\vec{p_j} + \vec{d_e} - \vec{p_i})\vec{s}^T | \forall e = \langle L_i, L_j \rangle \text{due to } w\}, \Pi_{k=1}^{n} b_k). \tag{4}$$

In the Jacobi example, the coalesced distance of the flow dependence for array *temp* equals $N-2$, which is less than $\Pi_{k=1}^{n} b_k = (N-2)^2$. Thus, the memory-space cost for the write reference to array *temp* equals $N-2$.

Next, we determine the total memory-space cost for all write references to local array regions. To simplify the discussion, we make the following assumption, under which the total memory-space cost for all write references is equal to the summation of the memory-space cost for each write reference.

Assumption 2 *We assume that the local array regions written by different write references do not overlap, and that all local array regions remain live until the end of the T loop (in Figure 4(a)).*

We want to formulate the memory space cost minimization problem as a network flow problem which can be solved in polynomial time [33]. There exist two obstacles: (1) the operator *min* in Formula (4) makes the minimization problem complex, and (2) non-constant dependence distance vectors make computing the shortest path distance difficult in network flow algorithm [1].

To overcome the above two obstacles, we first replace any edge which has a non-constant dependence distance vector, say $\vec{d_e}$, by two edges whose dependence distance vectors are constant. Recall that the coalesced distance $\vec{d_e}\vec{s}^T$ is a function of the index vector $\vec{i}$ of the loop nest L_i which contains the dependence source. We find two iteration vectors $\vec{i_0}$ and $\vec{i_0'}$ within the iteration domain such that the coalesced distance is minimum at $\vec{i_0}$ and maximum at $\vec{i_0'}$. We replace the edge of distance vector $\vec{d_e}$ by two edges and mark them by the two constant-dependence distance vectors corresponding to $\vec{i_0}$ and $\vec{i_0'}$ respectively. (In Appendix C, we prove that this replacement does not change the result of memory minimization.) As a common example, consider a non-constant distance vector whose elements are affine functions of $\vec{i}$. The coalesced distance is of course also an affine function of $\vec{i}$, say $\Sigma_{k=1}^{n} a_k i_k + a_0$. To find an index vector for which the coalesced distance is minimum, we simply let i_k equal the lower bound of L_{ik} if $a_k \geq 0$ and the upper bound of L_{ik} if $a_k < 0$. To find an index vector for which the coalesced distance is maximum, we do the opposite.

Let $G' = (V', E')$ be the modified loop dependence graph after replacing the non-constant dependence distance vectors. We further make the following assumption,

Assumption 3 *We assume that in G' the sum of the absolute values $|d_k|$ of all data dependence distances at loop level k are less than the trip count b_k.*

Under Assumptions 1 to 3, the following theorem formalizes our problem of minimizing the overall memory-space cost.

Theorem 1 *Given the m loop nests in Figure 4(a) and the revised loop dependence graph G', let τ_i be the number of write references to local array regions in each loop nest L_i. Let e represent an arbitrary data dependence in E', say from L_i to L_j, which may be a flow, anti-,*

or output dependence. Let e_1 represent an arbitrary flow dependence in E', say from L_i to L_j, for a local array region. Under loop shifting and loop fusion, the minimum memory-space cost is

$$min(\Sigma_{i=1}^{m}\Sigma_{k=1}^{\tau_i}\vec{M_{i,k}}\vec{s}^T) \tag{5}$$

subject to

$$(\vec{p_j} + \vec{d_e} - \vec{p_i})\vec{s}^T \geq 0, \forall e \in E', and \tag{6}$$

$$\vec{M_{i,k}}\vec{s}^T \geq (\vec{p_j} + \vec{d_{e_1}} - \vec{p_i})\vec{s}^T, \forall e_1 \in E', 1 \leq k \leq \tau_i. \tag{7}$$

Proof See Appendix C. □

Given the shifting vectors, the expression $\vec{M_{i,k}}\vec{s}^T$ in constraint (7) defines the maximum coalesced distance among all flow dependences originating from a write reference $w_{i,k}$ in L_i which writes to a local array region. According to Lemma 4, constraint (6) guarantees legal loop fusion after shifting and coalescing. The objective function (5) states the minimum total memory required for all static write references. A separate paper [33] describes how the problem formulated above can be reduced to a network-flow problem, without loss of optimality, which can be solved in polynomial time.

4 The Controlled SFC Scheme

A fusion scheme that minimizes the total memory requirement does not necessarily result in the minimum cache misses, since bringing more memory references into a loop body may potentially increase cache misses. Hence, we present a controlled-fusion scheme in this section.

Following [16], we classify all cache misses into three classes: *compulsory misses*, *capacity*

misses and *conflict misses*. Compulsory misses are those cache misses which are incurred when memory locations are first visited during program execution. Compared to the other two classes of cache misses, compulsory misses usually have much less impact on the overall program performance. Therefore we omit them in our study even though the SFC can potentially reduce compulsory misses because of the reduced array sizes. Capacity misses are due to limited cache size. These are noncompulsory cache misses which will occur even if the cache is fully associative. Conflict misses, on the other hand, are due to limited set associativity which causes different data items to interfere on the same cache set. Conflict misses could have been avoided if the cache were fully associative.

Our controlled SFC scheme is mainly aimed at reducing capacity misses. The number of capacity misses depends on both the cache size and the *reuse distances* in the given program. For a pair of consecutive references to the same data element, let δ be the number of distinct data elements accessed between such a pair. On a fully-associative cache with an LRU cache replacement policy, if δ exceeds the cache size, then the second reference will cause a capacity miss. The number δ is called *squeezed reference distance* by Gu *et al.* [14] and *reuse distance* by Ding and Kennedy [7]. Therefore, unless the cache size is greater than the maximum reuse distance, capacity misses will occur. Generally speaking, a smaller average reuse distance implies a better data locality in the program.

Loop fusion in SFC may either increase or decrease the reuse distance between a pair of memory operations on the same location. If the pair were in different loop nests before fusion, loop fusion tends to reduce their reuse distance. However, if the pair were in the same loop nest before fusion, loop fusion may introduce new memory operations between them which access new memory locations. The reuse distance between this pair will then

increase. Array contraction tends to reduce reuse distances. This is because certain memory operations which previously access distinct locations may access the same location after the contraction. If the reuse distance between two memory operations was greater than the cache size prior to array contraction, then the second memory operation would be a capacity miss. Suppose the reuse distance after array contraction is less than the cache size, then the second memory operation will no longer generate a capacity miss. Array contraction, therefore, can potentially reduce capacity misses. As far as registers are concerned, loop fusion tends to increase the pressure on register allocation due to the increased loop body size, while array contraction itself has less impact on register pressure. Excessive fusion may cause a loss in data locality and register spills. To avoid this, we selectively apply loop fusion such that the maximum reuse distance does not exceed the cache capacity and the data size of the loop body does not exceed the register limit after loop fusion and array contraction.

In [19], Kennedy and McKinley prove that fusion to maximize data locality is an NP-hard problem. Their work does not consider loop shifting or array contraction. Adding these two transformations into consideration further complicates the overall problem. Nonetheless, we believe the best loop candidates for fusion are those whose fusion can result in a maximal array contraction, subject to the conditions that the cache capacity is not exceeded by the maximum reuse distance and the register limit is not exceeded by the loop body. Under such a strategy, we devise an algorithm to greedily find fusion candidates which can result in a maximal array contraction in each step. This algorithm, named *controlled Fusion*, is used in the last stage of controlled SFC and is presented next.

Input: (1) a collection of m loop nests, L_i, as shown in Figure 4(a);
(2) shifting factors solved from the problem defined in Theorem 1;
(3) the sizes of local arrays, before and after maximal contraction;
(4) the cache size and the number of available registers.
Output: A set of partitions, $\mathcal{P}$, over the given loop nests.
Procedure:
Sort the contractable arrays, starting with the one which gives the maximum array size reduction.
Initialize the partition set $\mathcal{P}$ to ϕ.
for (each contractable array a in the sorted order)
Let S be the set of loop nests in which a is accessed.
Find the partitions in current $\mathcal{P}$ which overlap with S.
Let M be the union of all such partitions and S.
if ($num\_ref\_size(M)$ is smaller than the cache size **and** $num\_reg(M)$ is smaller than the number of available registers) **then**
remove the union members of M from $\mathcal{P}$ and add M to $\mathcal{P}$.
end if
end for
Add the set $\{L_i\}$ to $\mathcal{P}$ for all L_i not belonging to any element in $\mathcal{P}$.

Figure 5: Algorithm 1: controlled fusion

4.1 Controlled Fusion

Figure 5 shows our controlled fusion algorithm. It utilizes two functions. The function $num\_ref\_size(S)$ estimates the maximum reuse distance, under the assumptions that all the loops in S are fused together. Furthermore, all local array regions in S are assumed to have been contracted to the minimum size according to Theorem 1 which is applied to the loop nests in S only. Function $num\_reg(S)$ estimates the number of registers required to execute *a single iteration* of the innermost loop, assuming all the loops in S are fused together. Similar to previous work [30], we estimate the number of required registers before fusion as the number of distinct array elements accessed in a single iteration of the innermost loop. After fusion, instead of recounting, we follow previous work [31] and estimate the required number of registers by adding the numbers in the individual loops before fusion. (Detailed knowledge about the compiler back-end may allow one to estimate the required registers more precisely. This is because one can better predict how the machine code will be generated for the loop body and what register allocation algorithm will be used. However, based on our experiments and work of others [30, 31], the estimation method used here seems adequate.)

W_i: the total array data size of loop nest L_i, $1 \leq i \leq m$, in Figure 4(a).
P_j: the j-th member of $\mathcal{P}$ computed in Algorithm 1 (Figure 5).
F_j: the total array data size of P_j, after all loops in P_j are fused into one and array contraction is performed on P_j.
C_b and C_s: the cache line size and the cache size respectively.

Figure 6: Definition of some terms for cache miss estimation

Take `Jacobi` (in Figure 1) as an example. After solving the problem in Theorem 1, we have $\vec{p_1} = (0, 0)$ and $\vec{p_2} = (1, 0)$. The array *temp* is the only contractable array. Following the steps in Algorithm 1, we have $\mathcal{P} = \phi$ and $S = \{L_1, L_2\}$. No partition in $\mathcal{P}$ overlaps with S. We have $M = \{L_1, L_2\}$, *num_ref_size*$(M) = 3 * N - 2$ and *num_reg*$(M) = 7$. If $3 * N - 2$ is smaller than the cache size and the number of available registers is not smaller than 7, $\mathcal{P} = \{\{L_1, L_2\}\}$. Otherwise, $\mathcal{P} = \{\{L_1\}, \{L_2\}\}$.

Let n_a be the number of contractable arrays. The complexity of controlled SFC is itemized below. It takes time $O(n(V + E)^3)$ to compute the shifting factors [33]. It takes time $O(n_a log n_a)$ to sort the contractable arrays. Function *num_ref_size(M)* takes $O(n_a)$ time [33], and Function *num_reg(M)* takes time $O(m)$. The top-level **for** loop takes time $O(m^2 n_a(n_a + m))$. In summary, the total complexity is $O(max(n(V + E)^3, m^2 n_a(n_a + m)))$.

4.2 Effect on Capacity Misses

To illustrate the potential impact of controlled SFC on capacity misses, we define certain terms as in Figure 6 and assume the LRU cache replacement policy, which is a common policy in practice. We also assume cache line spatial locality is fully exploited for all loop nests. According to a study conducted by McKinley and Teman for a set of scientific programs in [25], most cache misses are *inter-nest* capacity misses, i.e. capacity misses that occur when program control proceeds from one loop nest to the next. We pessimistically assume that no cached-data reuse exists across different loop nests L_i in Figure 4(a), but we optimistically

assume that within each L_i, the reuse distances are all less than the cache size. Under these assumptions, using the notations in Figure 6, the number of capacity misses in the original code equals $\frac{\Sigma_{i=1}^{m} W_i}{C_b} * ITMAX$.

For the code after controlled SFC, we first consider the best case in which the total data size after applying controlled SFC is no greater than the cache size. The number of capacity misses will equal the total data size divided by the cache line size. If, however, the total data size after controlled SFC still exceeds the cache size, the number of capacity misses can be potentially greater. Again, we pessimistically assume that no cached-data reuse exists across different partitions, but we optimistically assume that the reuse distances are all less than the cache size within the same partition, the number of capacity misses is then upper-bounded by $\frac{\Sigma_{j=1}^{\| \mathcal{P} \|} F_j}{C_b} * ITMAX$, where $\| \mathcal{P} \|$ denotes the number of partitions in $\mathcal{P}$. Note that controlled SFC guarantees that the reuse distances within each partition P_i are no greater than the cache size if that partition contains more than one original loop nest.

Since $F_j \leq \Sigma_{L_i \in P_j} W_i$, even the upper bound of capacity misses after controlled SFC is no greater than capacity misses in the original code. If the data size after controlled SFC is within the cache size, then the reduction in capacity misses is very significant. In the cases between the best and the worst, the analysis of the exact number of capacity misses requires further assumptions, which is beyond the scope of this paper.

4.3 Experiments

We have implemented controlled SFC in a research Fortran77 compiler, Panorama [15, 33], and applied it to twenty test programs. Both the description of these test programs and

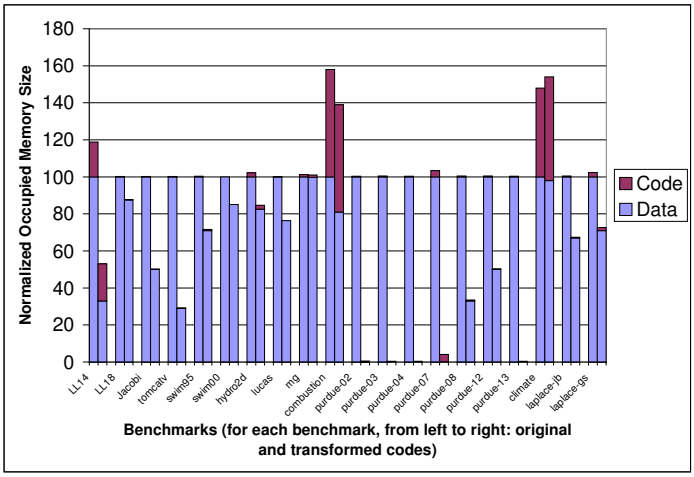


Figure 7: Memory sizes before and after transformation

details of the experimental results are given in Appendix D. In this subsection, we briefly summarize the results.

Figure 7 compares the code size and the data size before and after applying controlled SFC on an MIPS R10K processor. The data size shown for each original program is normalized to 100. The actual data size varies greatly for different benchmarks, which is listed in Appendix D. The arithmetic mean of the reduction rate, counting both the data and the code, is 51%. For several small benchmarks, the reduction rate is almost 100%. For each of the benchmarks, we examine three versions of the code, i.e. the original one, the one after loop fusion but before array contraction, and the one after array contraction. Among the tested benchmarks, only `combustion, purdue-07 and purdue-08` fit the program model in previous work [11]. In those cases, the previous algorithm [11] will derive the same result as ours. Therefore, there is no need to list them again.

We use the Cheetah cache simulator in SimplesScalar 3.0 suite [4] to help evaluate the

Table 1: Average reuse distances, reference counts and cache miss rates

		Original	Fusion	Contraction
Jacobi	avg. reuse distance	5208	3039	766
	reference count	10594956	9417607	10573205
	miss rate	0.113	0.064	0.029
tomcatv	avg. reuse distance	2919	1529	222
	reference count	122994043	125325245	125972343
	miss rate	0.052	0.018	0.005
swim95	avg. reuse distance	3628	3177	2090
	reference count	31063966	31740919	32031613
	miss rate	0.082	0.067	0.054

(Simulation results are obtained by using cache size 32KB and line size 16 bytes with LRU replacement. The programs are compiled by `gcc` with the "-O3" optimization level.)

effect of controlled SFC on data locality. Table 1 shows the average reuse distances for three programs. It also lists the total number of memory references and the cache miss rates. We use reduced data size for simulation in order to reduce the simulation time. From Table 1, we see that the average reuse distances after array contraction are lower than those after applying loop fusion alone. The latter are lower than those in the original programs.

To further evaluate the effectiveness of array contraction, we measured the performance on a MIPS R10K processor in an SGI Origin 2000 multiprocessor environment. For all versions of the benchmarks, we use the native Fortran compiler on an SGI Origin 2000 multiprocessor to produce the machine codes. We use the optimization flag "-O3", with the following adjustments to get as much performance as possible for all versions of codes. We switch off prefetching for `laplace-jb`, software pipelining for `laplace-gs` and loop unrolling for `purdue-03`. For `swim95` and `swim00`, the native compiler fails to insert prefetch instructions in the innermost loop body after memory reduction. We manually insert prefetch instructions into the three key innermost loop bodies, following exactly the same prefetching patterns used by the native compiler for the original codes. Figure 8 shows the normalized execution time of the twenty benchmark programs using the full data size. These measured

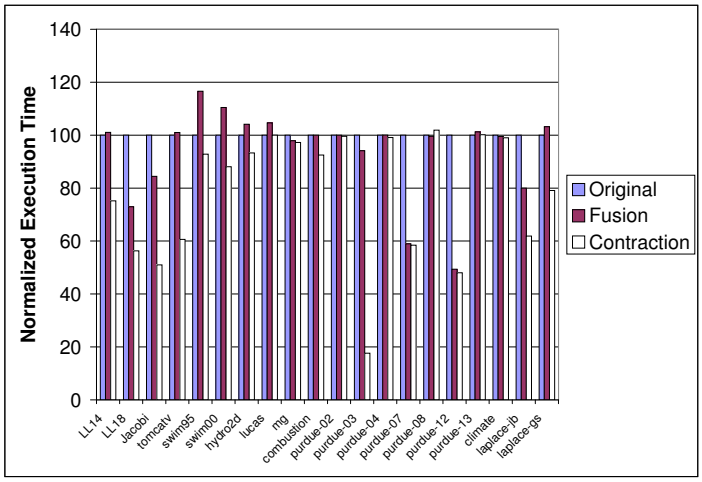
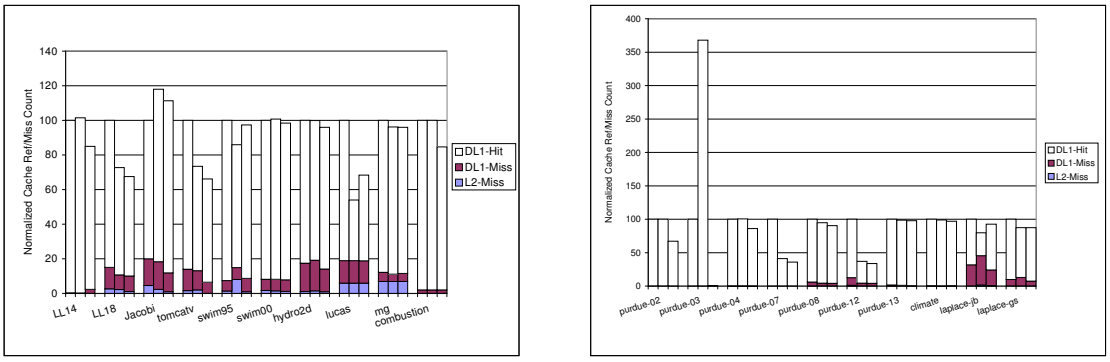


Figure 8: Performance comparision with the original inputs

results show that controlled SFC speeds up program execution by a factor of 1.40 over the original codes and a factor of 1.25 over the codes transformed by loop fusion only. We use the **perfex** [20] package to measure the number of misses on the L1 data cache and the L2 unified cache. Figure 9 compares such statistics, where the total reference counts in the original codes are normalized to 100. The results show that controlled SFC either reduces the total number of loads and stores (by array scalarization) or the number of cache misses (by partial contraction), often both.

4.4 Discussions

The programmer or the compiler can reduce conflict misses significantly by a careful choice of the array column sizes and the starting addresses of different arrays [28, 29]. After array contraction, such techniques can still be applied in order to reduce conflict misses. In fact, if an array is contracted to a single dimension or to a scalar, interferences between different array columns at the same cache line naturally disappear. On the other hand, when loop



(Original, Fusion and Contraction are from left to right for each benchmark)

Figure 9: Cache statistics with the original inputs

fusion is performed to increase the opportunity for array contraction, the number of array elements referenced in a loop nest may be increased, which in turn may increase conflict misses. However, our experiments show that the benefit of reduced capacity misses tend to outweight the penalty of any increased conflict misses.

Controlled SFC may have an impact on software pipelining. Loop fusion increases the length of the loop body. This can potentially increase the number of simultaneous operations. On the other hand, loop fusion also increases resource requirement. For example, it may increase the number of registers required to allow simultaneous operations. Consequently, the minimum initiation interval (MII) [2] may be increased, offsetting the potential gain in parallelism. Furthermore, both loop fusion and array contraction may introduce additional data dependences in the loop body, which may also increase the MII. All these factors may affect the software pipelining schedule in a way which is difficult to predict when the compiler transforms the program with a representation close to the source code level.

5 Conclusion and Future Work

In this paper, we have formulated a memory-requirement minimization problem under loop shifting, loop fusion and array contraction. We have also presented the controlled SFC scheme to avoid over-fusing loops. We have studied the impact of our technique on cache misses through a combination of analysis, simulation and measurement. The results demonstrate that the controlled SFC can effectively reduce cache misses and improve execution speed. More details of the experiments are reported in the Appendices which will accompany this paper as supplemental materials on the digital library of IEEE Computer Society.

The program model presented in this paper imposes a number of requirements on the given loop nesting. For example, the loop bounds are required to be loop-nest invariants. In our future work, we shall investigate extensions to our current program model, e.g., extensions that cover loop bounds which are affine functions of the enclosing loops.

Currently, most program transformations intended for cache-performance enhancement, including our controlled SFC, are performed with a representation close to the source code level. In the future, we believe such transformations are best performed at a lower level representation, where the compiler can have better information on the impact of the transformations on low level technique such as software pipelining and register allocation.

Acknowledgement

This work is sponsored in part by National Science Foundation through grants CCR-950254, CCR-9975309, ITR/ACR-0082834 and CCR-0208760. The authors thank the reviewers for

their careful reviews and useful suggestions. In particular, the critical review by one of the reviewers has resulted in a more complete and robust theoretical development in this paper.

References

[1] R. Ahuja, T. Magnanti, and J. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1993.

[2] Vicki Allan, Reese Jones, Randall Lee, and Stephen Allan. Software pipelining. *ACM Computing Surveys*, 27(3):367–432, September 1995.

[3] David Bacon, Susan Graham, and Oliver Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, December 1994.

[4] Doug Burger and Todd Austin. The SimpleScalar tool set, version 2.0. Technical Report TR-1342, Department of Computer Sciences, Univ. of Wisconsin, Madison, June 1997.

[5] Béatrice Creusillet and Francois Irigoin. Interprocedural array region analyses. *International Journal of Parallel Programming*, 24(6):513–546, December 1996.

[6] Alain Darte. On the complexity of loop fusion. In *Proceedings of International Conference on Parallel Architecture and Compilation Techniques*, pages 149–157, Newport Beach, California, October 1999.

[7] Chen Ding and Ken Kennedy. Improving effective bandwidth through compiler enhancement of global cache reuse. In *Proceedings of 2001 International Parallel and Distributed Processing Symposium*, San Francisco, California, April 2001.

[8] Paul Feautrier. Array dataflow analysis. In *Compiler Optimizations for Scalable Parallel Systems Languages 2001: 173-220, Lecture Notes in Computer Science 1808 Springer 2001, ISBN 3-540-41945-4.*

[9] Paul Feautrier. Array expansion. In *Proceedings of the 1988 ACM International Conference on Supercomputing*, pages 429–441, July 1988.

[10] Paul Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, January 1991.

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

[12] Guang R. Gao, Russell Olsen, Vivek Sarkar, and Radhika 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.

[13] Thomas Gross and Peter Steenkiste. Structured dataflow analysis for arrays and its use in an optimizing compiler. *Software-Practice and Experience*, 20(2), February 1990.

[14] Junjie Gu, Zhiyuan Li, and Gyungho Lee. An evaluation of the potential benefits of register allocation for array references. In *Workshop on Interaction between Compilers and Computer Architectures in conjunction with the HPCA-2*, San Jose, California, February 1996.

[15] Junjie Gu, Zhiyuan Li, and Gyungho Lee. Experience with efficient array data flow analysis for array privatization. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 157–167, Las Vegas, NV, June 1997.

[16] John Hennessy and David Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, 1996.

[17] Mahmut Kandemir, Alok Choudhary, J. Ramanujam, and Prithviraj Banerjee. A matrix-based approach to global locality optimization. *Journal of Parallel and Distributed Computing*, 58(2):190–235, 1999.

[18] Ken Kennedy. Fast greedy weighted fusion. In *Proceedings of the 2000 International Conference on Supercomputing*, Santa Fe, New Mexico, May 2000.

[19] Ken Kennedy and Kathryn 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.

[20] James Laudon and Daniel Lenoski. The SGI Origin: a ccNUMA highly scalable server. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 241–251, Denver, CO, June 1997.

[21] Vincent Lefebvre and Paul Feautrier. Automatic storage management for parallel programs. *Parallel Computing*, 24(3-4):649–671, May 1998.

[22] Amy W. Lim, Shih-Wei Liao, and Monica 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.

[23] Vincent Loechner, Benot Meister, and Philippe Clauss. Precise data locality optimization of nested loops. *The Journal of Supercomputing*, 21(1):37–76, 2002.

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

[25] Kathryn S. McKinley and Olivier Teman. Quantifying loop nest locality using SPEC'95 and the Perfect benchmarks. *ACM Transactions on Computer Systems*, 17(4), November 1999.

[26] A. Gaber Mohamed, Geoffrey C. Fox, Gregor von Laszewski, Manish Parashar, Tomasz Haupt, Kim Mills, Ying-Hua Lu, Neng-Tan Lin, and Nang-Kang Yeh. Applications benchmark set for Fortran-D and High Performance Fortran. Technical Report CRPS-TR92260, Center for Research on Parallel Computation, Rice University, June 1992.

[27] John Rice and J. Jing. Problems to test parallel and vector languages. Technical Report CSD-TR-1016, Department of Computer Science, Purdue University, 1990.

[28] Gabriel Rivera and Chau-Wen Tseng. Eliminating conflict misses for high performance architectures. In *Proceedings of the 1998 ACM International Conference on Supercomputing*, pages 353–360, Melbourne, Australia, July 1998.

[29] Gabriel Rivera and Chau-Wen Tseng. A comparison of compiler tiling algorithms. In *Proceedings of the Eighth International Conference on Compiler Construction*, Amsterdam, The Netherlands, March 1999.

[30] Vivek Sarkar. Optimized unrolling of nested loops. In *Proceedings of the ACM International Conference on Supercomputing*, pages 153–166, Santa Fe, NM, May 2000.

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

[32] Yonghong Song, Rong Xu, Cheng Wang, and Zhiyuan Li. Performance enhancement by memory reduction. Technical Report CSD-TR-00-016, Department of Computer Science, Purdue University, 2000. Also available at http://www.cs.purdue.edu/homes/songyh/academic.html.

[33] Yonghong Song, Rong Xu, Cheng Wang, and Zhiyuan Li. Data locality enhancement by memory reduction. In *Proceedings of the 15th ACM International Conference on Supercomputing*, Naples, Italy, June 2001.

[34] Waibhav Tembe and Santosh Pande. Data I/O minimization for loops on limited on-chip memory processors. *IEEE Transactions on Computers*, 51(10):1269–1280, October 2002.

[35] William Thies, Frédéric Vivien, Jeffrey Sheldon, and Saman Amarasinghe. A unified framework for schedule and storage optimization. In *Proceedings of 2001 ACM Conference on PLDI*, pages 232–242, Snowbird, Utah, June 2001.

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

Appendix A: Proof of Lemma 1

Suppose the given set of shifting vectors do not already allow legal fusion without loop coalescing. The loop dependence graph (LDG) must be a DAG if we ignore the self cycles. Thus we can traverse the LDG in a topological order. At each node L_i, we find the incoming edge whose distance vector $\vec{d}$ under the given shifting vectors is the lexicographical minimum among all incoming edges to the same node. Suppose $\vec{d} = (0, \ldots, 0, d_k, \ldots)$, where the first non-zero component is the kth component $d_k < 0$. We let $\vec{q} = (0, \ldots, 0, -d_k + 1, -(-d_k + 1)b_{k+1}, 0, \ldots, 0)$, where the kth component of $\vec{q}$ is $-d_k + 1$. We then add $\vec{q}$ to the shifting vector $\vec{p_i}$ of L_i. The distance vectors of all incoming edges to L_i are now lexicographically positive. Furthermore, the coalesced distance of each incoming edge remains unchanged because the difference between the old and the new coalesced distance equals $\vec{q}\vec{s}^T = 0$.

Appendix B: Proof of Lemma 2

Suppose $\vec{u} \succ \vec{0}$. Since $abs(\vec{u}) < \vec{b}$, we have $\vec{u} \geq \vec{v} = (0, \ldots, 0, 1, 1 - b_{k+1}, \ldots, 1 - b_n)$, $1 \leq k \leq n$, assuming the first non-zero element in $\vec{u}$ is the kth element. Since $\vec{u} - \vec{v} \geq \vec{0}$, we have $\vec{u}\vec{s}^T - \vec{v}\vec{s}^T = (\vec{u} - \vec{v})\vec{s}^T \geq 0$, hence $\vec{u}\vec{s}^T \geq \vec{v}\vec{s}^T$. Since $\vec{v}\vec{s}^T = s_n > 0$, we have $\vec{u}\vec{s}^T > 0$.

Similarly, we can prove that $\vec{u} \prec \vec{0} \Rightarrow \vec{u}\vec{s}^T < 0$. Therefore, if $\vec{u}\vec{s}^T > 0$, then $\vec{u} \succ \vec{0}$ must be true.

Appendix C: Proof of Theorem 1

From Lemma 5, the objective function of the memory-space cost minimization should be

$$min(\Sigma_{i=1}^{m}\Sigma_{k=1}^{\tau_i} min(\vec{M_{i,k}}\bar{s}^T, \Pi_{t=1}^{n} b_t)), \tag{8}$$

under constraints (6) and (7). We define this problem (the objective function (8) and the constraints (6) and (7)) as **Problem 0** and the problem in Theorem 1 as **Problem 1**.

It is clear that there exist optimal solutions for **Problem 0** and for **Problem 1**. This is because constraints (6) and (7) guarantee $\vec{M_{i,k}}\bar{s}^T \geq 0$, which bounds the objective functions (5) and (8) from below. Thus, to prove Theorem 1, we only need to prove the following two claims.

- **Claim C.1:** For any optimal solution of **Problem 0**, we can find a solution for **Problem 1** with the same objective function value. Conversely, for any optimal solution of **Problem 1**, we can find a solution for **Problem 0** with the same objective function value.

- **Claim C.2:** In Section 3.4, we replaced each dependence edge which has a non-constant distance vector by two edges whose distance vectors are constant. One vector has the maximum coalesced distance and the other with the minimum coalesced distance. We claim that any edge we replaced would have satisfied constraints (6) and (7) under an optimal solution of shifting vectors. Therefore, its removal does not affect the optimal objective function value in Formulas (5) and (8).

We prove Claim C.1 first. Without any loss of generality, we can assume that G' has a single *connected component* when the directions of edges are ignored. This is because, if G' viewed

as an undirected graph contains two or more connected components, then the shifting vectors can be solved for each component independent of the other components.

For any optimal solution of **Problem 0** over G', we construct a new graph G_1 which is a subset of G' obtained by removing the edges whose coalesced distances, under the optimal shifting vectors, do not equal zero. We have the following lemma.

Lemma 6 *There exists an optimal solution of* **Problem 0** *over* G' *such that its corresponding graph* G_1*, when the edge directions are ignored, has a single connected component.*

Proof Suppose G_1, when the edge directions are ignored, contains more than one connected component. Take any component in G_1, say t_1. This component may contain flow-dependence sources for certain local array regions and flow-dependence destinations for others. Suppose we further shift all nodes in t_1 by a shifting factor -1. This is equivalent to further shifting the innermost loop by -1 after optimal loop shifting. This is always legal because all dependences with either the source or the destination (but not both) in t_1 have positive dependence distances. The shifting will have no impact on data dependences whose sources and destinations are both in t_1.

The local array regions which have flow-dependence sources (but not all destinations) in t_1 may have their minimum memory requirement increased. Let κ_1 represent the increased memory requirement for such local array regions. The local array regions which have flow-dependence destinations (but not all the sources) in t_1 may have their minimum memory requirement reduced. Let κ_2 represent the decreased memory requirement for such local array regions. Similarly, if we shift all nodes in t_1 in G' by 1, we can let κ_3 and κ_4 represent

the memory requirement increase and decrease respectively, for their corresponding local array regions.

Consider local array regions which have flow-dependence sources in t_1 but not all the destinations. As stated above, shifting by -1 may potentially increase the memory requirement by κ_1, and shifting by 1 may decrease memory requirement by κ_4. Now, for any local array region whose flow-dependence source is in t_1 but whose flow-dependence destinations are not all in t_1, we have the following two scenarios:

- Shifting by 1 reduces its memory requirement by 1. Then, shifting by -1 may increase its memory requirement by 1, or the increase may be just 0 if the coalesced dependence distance is equal to $\Pi_{t=1}^{n} b_t$, according to Formula (8).

- Shifting by 1 does not change its memory requirement. This means two possibilities. The first is that the destination node corresponding to the maximum coalesced dependence distance is inside t_1 and the coalesced dependence distances corresponding to those destinations outside t_1 are *smaller* than that maximum coalesced distance. The second possibility is that the maximum coalesced dependence distance is already greater than $\Pi_{t=1}^{n} b_t$. In either case, shifting by -1 also does not change the memory requirement.

From the discussions above, for the local array region under consideration, the decreased memory requirement is equal to or greater than the increased memory requirement. Considering all such local array regions, we have $\kappa_4 \geq \kappa_1$. Similarly, $\kappa_2 \geq \kappa_3$ holds. Because our solution is optimal, both $\kappa_1 - \kappa_2 \geq 0$ and $\kappa_3 - \kappa_4 \geq 0$ hold. Thus, we have $\kappa_1 = \kappa_2 = \kappa_3 = \kappa_4$, which implies that we can further shift all the nodes in t_1 in G' either by -1 or by 1 without

changing the objective function value. The updated shifting vectors $\vec{p_i}$ will remain optimal. We continue to further shift until one of the original edges between t_1 and other nodes in G' has $(\vec{p_j} + \vec{d_e} - \vec{p_i})\vec{s}^T = 0$. At this time, the number of connected components in G_1 is reduced by 1 with a new set of optimal shifting vectors.

We repeat the above process until only one connected component remains in G_1. Thus, we have transformed the original optimal solution to a new optimal solution such that its associated G_1 has a single connected component when edge directions are ignored. □

Lemma 7 below is the counterpart of Lemma 6 for **Problem 1**. Its proof is nearly identical to that of Lemma 6. The phrase "minimum memory requirement" in Lemma 6 is replaced by "maximum coalesced distance" in Lemma 7. There exist also certain subtle differences in the increase and the decrease of these numbers. For completeness, we present the whole proof to Lemma 7.

Lemma 7 *There exists an optimal solution of* **Problem 1** *over G' such that its associated G_1, when the edge directions are ignored, has a single connected component.*

Proof Suppose G_1 contains more than one connected component. Take any component in G_1, say t_1.

Imagine that we further shift all nodes in t_1 in G' by -1. The local array regions which have flow dependence sources (but not all destinations) in t_1 may have maximum coalesced dependence distance increased. We let κ_1 denote this increase. The local array regions which have some flow dependence destinations (but not the source) in t_1 may have maximum coalesced dependence distance reduced. We let κ_2 denote this decrease. Similarly, imagine that we shift all nodes in t_1 in G' by 1, we let κ_3 and κ_4 represent the increase and decrease

in the maximum coalesced dependence distance respectively for their corresponding local array regions.

Consider local array regions which have the flow dependence sources in t_1 but not all the destinations. As stated in the above, shifting by -1 may potentially increase the maximum coalesced dependence distance by κ_1, and shifting by 1 may decrease the maximum coalesced dependence distance by κ_4. Now, for any local array region whose flow-dependence source is in t_1 but whose flow-dependence destinations are not all in t_1, we have the following two scenarios:

- Shifting by 1 reduces its maximum coalesced dependence distance by 1. Then, shifting by -1 will increase its maximum coalesced dependence distance by 1, according to Formula (5).

- Shifting by 1 does not change its maximum coalesced dependence distance. This means that the destination node corresponding to the maximum coalesced dependence distance is inside t_1 and the coalesced dependence distances corresponding to destinations outside t_1 are *smaller* than that maximum distance. Therefore, shifting by -1 also does not change the maximum coalesced dependence distance.

Hence, for this local array region, the decrease in the maximum coalesced dependence distances is equal to the increase in the maximum coalesced dependence distances among all flow dependences. Including all such local array regions, we have $\kappa_4 = \kappa_1$. Similarly, we can prove that $\kappa_2 = \kappa_3$ holds.

Because our solution is optimal, both $\kappa_1 - \kappa_2 \geq 0$ and $\kappa_3 - \kappa_4 \geq 0$ hold. These lead to $\kappa_1 = \kappa_2 = \kappa_3 = \kappa_4$, which implies that we can further shift all the nodes in t_1 in G' either

by -1 or by 1 without changing the objective function value. The updated shifting vectors $\vec{p_i}$ will remain optimal. The rest of the proof is identical to that of Lemma 6.

□

Under Assumption 3, we have the following lemma.

Lemma 8 *For any optimal solution of* **Problem 0** *(or* **Problem 1***) over G', if its associated G_1 has a single connected component when the edge directions are ignored, then the condition $(\vec{p_j} + \vec{d_e} - \vec{p_i})\vec{s}^T < \Pi_{t=1}^n b_t$ holds for all dependence distance edges in G'.*

Proof For any edge e from L_i to L_j in G', ignoring edge directions, there exists a simple path from L_i to L_j in G_1, say $(L_i, L_{q_1}, L_{q_2}, \ldots, L_{q_k}, L_j)$, where $k \geq 0$. We write L_i as L_{q_0} and L_j as $L_{q_{k+1}}$.

For any edge e_t between L_{q_t} and $L_{q_{t+1}}$, $0 \leq t \leq k$, if the edge direction is from L_{q_t} to $L_{q_{t+1}}$, let $r_t = 1$. Otherwise, let $r_t = -1$. For edge e_t, we have

$$(\vec{p_{q_{t+1}}} + r_t\vec{d_{e_t}} - \vec{p_{q_t}})\vec{s}^T = 0, 0 \leq t \leq k. \tag{9}$$

Adding together all $k+1$ equations represented by (9), we have $(\vec{p_j} + \Sigma_{t=0}^k r_t\vec{d_{e_t}} - \vec{p_i})\vec{s}^T = 0$. Therefore, $(\vec{p_j} + \vec{d_e} - \vec{p_i})\vec{s}^T = (-\Sigma_{t=0}^k r_t\vec{d_{e_t}} + \vec{d_e})\vec{s}^T$. Under Assumption 3, $(\vec{p_j} + \vec{d_e} - \vec{p_i})\vec{s}^T \leq (\vec{b} - \vec{1})\vec{s}^T < \Pi_{t=1}^n b_t$. □

Note that any feasible solution for **Problem 0** is also a solution for **Problem 1**, and *vice versa*. Based on Lemmas 6 and 8, for any optimal solution of **Problem 0** over G', we can find an optimal solution where the objective function value of (8) is equal to that of (5). Therefore, the optimal objective function value of (8) is equal to or greater than the optimal objective function value of (5). Similarly, for any optimal solution for **Problem 1** over G', based on Lemmas 7 and 8, we can find an optimal solution whose objective function

value for (5) is equal to that for (8). Therefore, the optimal objective function value for (5) is equal to or greater than the optimal objective function value of (8). Hence, **Problem 0** and **Problem 1** have the same optimal objective function value over G'. Claim C.1 is thus proved.

Next, we prove Claim C.2. Suppose dependence edge e has a non-constant dependence distance vector $\vec{d_e}$ which is replaced by two constant instances of $\vec{d_e}$, say $\vec{d_{e_1}}$ and $\vec{d_{e_2}}$, under certain instances of the iteration vector, such that $\vec{d_{e_1}}$ has the minimum coalesced dependence distance and $\vec{d_{e_k}}$ has the maximum coalesced dependence distance. We have

$$\vec{d_{e_1}}\vec{s}^T \leq \vec{d_{e_t}}\vec{s}^T \leq \vec{d_{e_k}}\vec{s}^T, 2 \leq t \leq k-1, \tag{10}$$

where $\vec{d_{e_t}}$ is an arbitrary instance of $\vec{d_e}$. In the following, we prove that adding any such distance vector $\vec{d_{e_t}}$ to either **Problem 0** or **Problem 1**, the optimal solution (of either problem) will not change. To prove this, we make the following derivations based on Inequality (10):

$$(\vec{p_j} + \vec{d_{e_1}} - \vec{p_i})\vec{s}^T \geq 0 \Leftrightarrow (\vec{p_j} + \vec{d_{e_t}} - \vec{p_i})\vec{s}^T \geq (\vec{d_{e_t}} - \vec{d_{e_1}})\vec{s}^T \Rightarrow (\vec{p_j} + \vec{d_{e_t}} - \vec{p_i})\vec{s}^T \geq 0 \tag{11}$$

Similarly, we make the following derivation:

$$\vec{M_{i,k}}\vec{s}^T \geq (\vec{p_j} + \vec{d_{e_k}} - \vec{p_i})\vec{s}^T \Rightarrow \vec{M_{i,k}}\vec{s}^T \geq (\vec{p_j} + \vec{d_{e_t}} - \vec{p_i})\vec{s}^T \tag{12}$$

Based on derivations (11) and (12), $\vec{d_{e_t}}$ satisfy both constraints (6) and (7). From (12), $\vec{d_{e_t}}$ does not affect the final solution for **Problem 0** and **Problem 1**. Hence, $\vec{d_{e_t}}$ can be omitted.

Combining Claims C.1 and C.2, we have shown that **Problem 0** and **Problem 1** have the same optimal objective function value over the loop dependence graph G. Furthermore,

given any optimal solution for one problem, we can derive an optimal solution for the other. Theorem 1 is thus proved.

Appendix D: Experimentation Details

We have implemented controlled SFC in a research Fortran77 compiler, Panorama [15, 33], and used it to find a number of test programs on which controlled SFC successfully reduces array sizes. (Although the question of how widely array contraction is applicable is of high empirical importance, such study requires a massive survey of open-source programs and is beyond the scope of this paper. We point out that, to our best knowledge, our test programs are much more extensive than reported in any previous papers on array contraction.)

We use the collected test cases to gather experimental results, from both simulation and real-machine measurement, concerning cache performance and execution speed. In this appendix, we describe the test cases and report experimentation details.

D.1 Test Cases

Table 2 lists the benchmarks used in our experiments, their descriptions and their input parameters. In this table, "m/n" represents the number of loops in the loop sequence (m) and the maximum loop nesting level (n). Note that the array size and the iteration counts are chosen arbitrarily for `LL14`, `LL18` and `Jacobi`. To distinguish the two versions of `swim` in SPEC95 and SPEC2000, respectively, we denote the SPEC95 version as `swim95` and the SPEC2000 version as `swim00`. Program `swim00` is almost identical to `swim95` except for its larger data size. For `combustion`, we change the array size (N1 and N2) from 1 to 10, to make the execution time last for at least several seconds. Programs `climate`,

Table 2: Test programs

Benchmark Name	Description	Input Parameters	m/n
LL14	Livermore Loop No. 14	N = 1001, ITMAX = 50000	3/1
LL18	Livermore Loop No. 18	N = 400, ITMAX = 100	3/2
Jacobi	Jacobi Kernel w/o convergence test	N = 1100, ITMAX = 1050	2/2
tomcatv	A mesh generation program from SPEC95fp	reference input	5/1
swim95	A weather prediction program from SPEC95fp	reference input	2/2
swim00	A weather prediction program from SPEC2000fp	reference input	2/2
hydro2d	An astrophysical program from SPEC95fp	reference input	10/2
lucas	A promality test from SPEC2000fp	reference input	3/1
mg	A multigrid solver from NPB2.3-serial benchmark	Class 'W'	2/1
combustion	A thermochemical program from UMD Chaos group	N1 = 10, N2 = 10	1/2
purdue-02	Purdue set problem02	reference input	2/1
purdue-03	Purdue set problem03	reference input	3/2
purdue-04	Purdue set problem04	reference input	3/2
purdue-07	Purdue set problem07	reference input	1/2
purdue-08	Purdue set problem08	reference input	1/2
purdue-12	Purdue set problem12	reference input	4/2
purdue-13	Purdue set problem13	reference input	2/1
climate	A two-layer shallow water climate model from Rice	reference input	2/4
laplace-jb	Jacobi method of Laplace from Rice	ICYCLE = 500	4/2
laplace-gs	Gauss-Seidel method of Laplace from Rice	ICYCLE = 500	3/2

`laplace-jb`, `laplace-gs` and all the Purdue set problems are from an HPF benchmark suite at Rice University [26, 27], which are available on-line. Except for `lucas`, all benchmarks are written in F77. We manually apply our technique to `lucas`, which is written in F90. We note that, prior to applying controlled SFC, loop interchange and array transpose are required for `tomcatv` and circular loop skewing is required for `swim`. They are all well-known transformations [17, 36]. Among 20 benchmark programs, our algorithm finds that the purdue-set programs, `lucas`, `LL14` and `combustion` do not require loop shifting in order to get their loops fused. For each of the benchmarks in Table 2, 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. For all benchmarks except `tomcatv`, `swim95` and `swim00`, we perform array contraction with the L1 cache as the target, i.e., we try to improve the L1 cache locality. For those three other programs, we target the L2 cache, which has a much bigger size. This is due to the large array column sizes and the large number of

(Data size for the original programs (unit: KB))

LL14	LL18	Jacobi	tomcatv	swim95	swim00	hydro2d
96	11520	19360	14750	14794	191000	11405
lucas	mg	combustion	purdue-02	purdue-03	purdue-04	purdue-07
142000	8300	89	4198	4198	4194	524
purdue-08	purdue-12	purdue-13	climate	laplace-jb	laplace-gs	-
4729	4194	4194	169	6292	1864	-

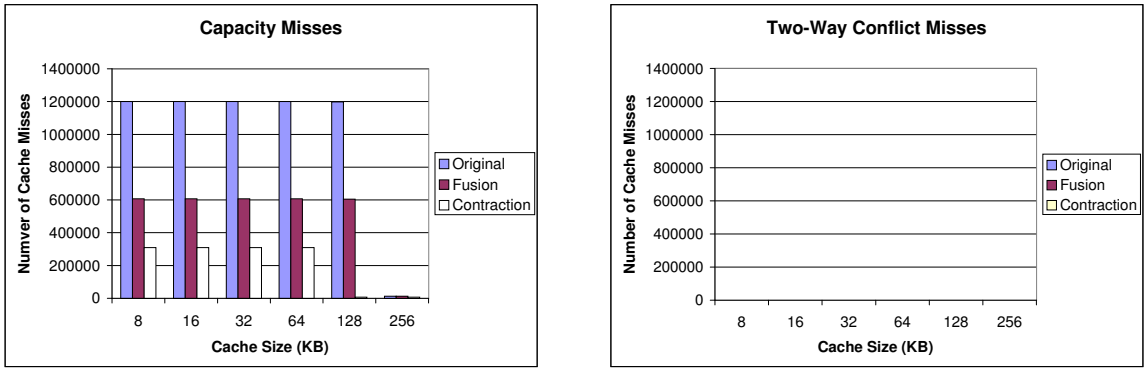
Table 3: Actual data sizes before and after transformation

arrays in those three programs. Unless all arrays are contracted from two dimensions to one dimension or scalars, the resulted maximum reuse distance after array contraction will not fit in the L1 cache.

Table 3 lists the actual data size before and after applying controlled SFC on an MIPS R10K processor in an SGI Origin 2000 multiprocessor environment. From Figure 7, for `mg` and `climate`, the memory requirement differs little before and after transformation. This is due to the small size of the contractable local array. For all other benchmarks, our technique reduces the memory requirement noticeably. Counting both the data and the code, the arithmetic mean of the reduction rate is 51%. Among the tested benchmarks, only `combustion, purdue-07 and purdue-08` fit the program model in previous work [11]. In those cases, the previous algorithm [11] will derive the same result as ours. So, there is no need to list those results.

D.2 Simulation Results

We use the Cheetah cache simulator in SimplesScalar 3.0 suite [4] to examine the cache behavior. Our simulated machine model contains a one-level cache with a line size of 16 bytes. We use the LRU cache replacement policy. The program is compiled by `gcc` with



(The number of conflict misses is less than 11 in all cases)

Figure 10: Cache statistics for Jacobi ($N = 110$, $ITMAX = 50$)

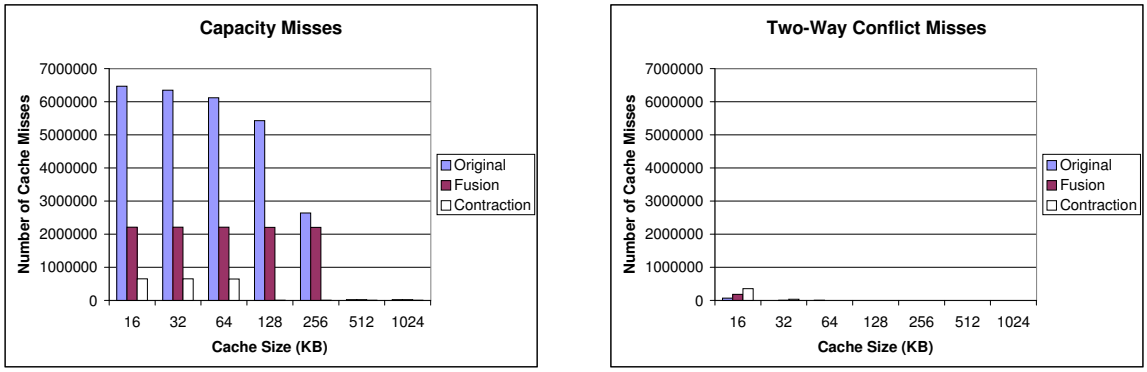


Figure 11: Cache statistics for `tomcatv` ($N = 80$, $ITMAX = 100$)

the optimization level "-O3". By using a simulator, we are able to separate capacity misses from conflict misses.

Unlike the execution on a real machine in Section D.3, we use the reduced data set size and smaller iterative-loop trip counts for simulation, in order to reduce the long simulation time. In this subsection, we pick three benchmarks to study their cache behavior, with arbitrarily-chosen inputs. We count the number of cache misses for the whole program which also contains some initialization and output codes.

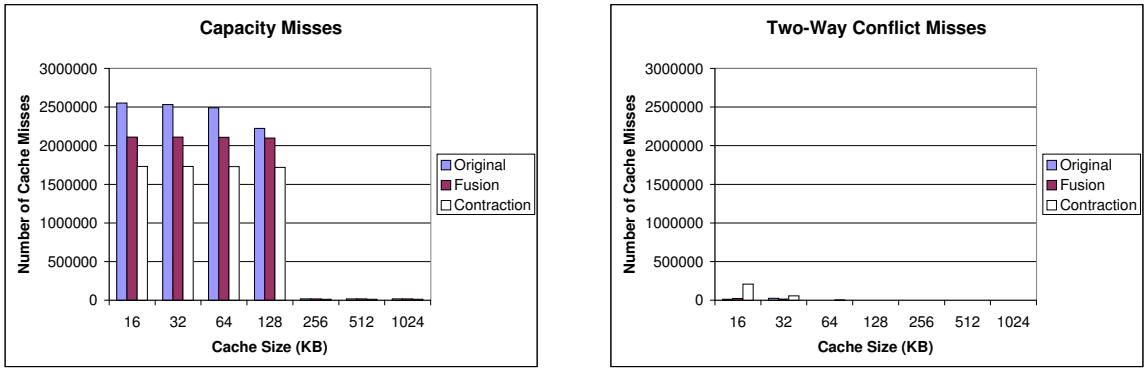


Figure 12: Cache statistics for `swim95` ($M = N = 64$, $ITMAX = 90$)

`Jacobi`

Figure 10 shows the cache miss results with $ITMAX = 50$ and $N = 110$. Each data element takes 8 bytes. The top subfigure shows that, for cache sizes 8KB to 64KB, loop fusion reduces capacity misses by half and array contraction further halves that number. For the cache size 128KB, array contraction makes both arrays fit in the cache, thus it maximally reduces the number of capacity misses. For the cache size 256KB, even the original arrays can all fit in the cache simultaneously, so the number of capacity misses is small for all three versions of codes. The other subfigure shows conflict misses, assuming a two-way associative cache. From this subfigure, we observe no significant negative effect of controlled SFC on conflict misses.

`tomcatv`

The original `tomcatv` has seven 2-D arrays accessed in 5 loop nests. The numbers of 2-D arrays referenced in these five loop nests are 6, 2, 5, 4 and 4 respectively. After fusion, the same seven 2-D arrays are accessed in the fused loop. After array contraction, five of the

2-D arrays are contracted to seven 1-D arrays. Figure 11 shows the results with the input $N = 80$ and $ITMAX = 100$.

`swim95`

The original `swim95` has thirteen 2-D arrays accessed in 2 loop nests. After fusion, the same thirteen 2-D arrays are accessed in the fused loop. After array contraction, four 2-D arrays (CU, CV, Z and H) are contracted to twelve 1-D arrays. Figure 12 shows the results with the input $M = N = 64$ and $ITMAX = 90$.

D.3 Execution Results on a Real Machine

To further evaluate the effectiveness of array contraction, we have measured the performance both on a Sun Ultrasparc machine and on one processor (MIPS R10K) in an SGI Origin 2000 multiprocessor. The results on these two machines are quite similar. We document the Ultrasparc results elsewhere [32] and report the MIPS R10K results as follows.

The MIPS R10K has a 32KB 2-way set-associative L1 data cache with a 32-byte cache line, and it has a 4MB 2-way set-associative unified L2 cache with an 128-byte cache line. The multiprocessor machine has a total of 16GB size of memory of which 1GB is local to the processor. It has 32 integer registers and 32 floating-point registers. The cache miss penalty is 9 CPU cycles for the L1 data cache and 68 CPU cycles for the L2 cache.

For all versions of the benchmarks, we use the native Fortran compiler on an SGI Origin 2000 multiprocessor to produce the machine codes. We use the optimization flag "-O3", with the following adjustments to get as much performance as possible for all versions of codes. We switch off prefetching for `laplace-jb`, software pipelining for `laplace-gs` and loop unrolling for `purdue-03`. For `swim95` and `swim00`, the native compiler fails to insert

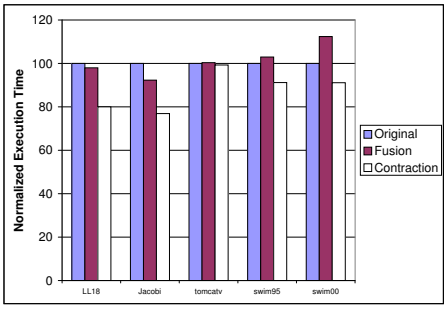


Figure 13: Performance comparision with $ITMAX = 1$

prefetch instructions in the innermost loop body after memory reduction. We manually insert prefetch instructions into the three key innermost loop bodies, following exactly the same prefetching patterns used by the native compiler for the original codes.

Execution Time with Original Inputs

From Figure 8 in the paper, one can see that loop fusion sometimes worsens the performance. As we discussed in the previous subsection, loop fusion may increase conflict misses in some cases. Nonetheless, the significant reduction in capacity misses after array contraction greatly outweights any increase in conflict misses introduced by loop fusion. From Figure 8, the codes after array contraction achieves an average speedup (using a geometric mean) of 1.40 over the original programs. The average speedup over the fusion codes is 1.25.

Cache Statistics

We also measured the reference count (dynamic load/store instructions), the number of misses on the L1 data cache, and the number of misses on the L2 unified cache. We use the **perfex** [20] package to get the cache statistics. Figure 9 in the paper compares such statistics, where the total reference counts in the original codes are normalized to 100. One special

example is `Jacobi`, in which the number of references is increased after array contraction, when compared with the original code. On the other hand, from Figure 9, the number of cache misses (especially L2 cache misses) is reduced significantly.

When arrays are contracted to scalars, register reuse is often increased. Figure 9 shows that the number of total references get decreased in most cases. However, in a few cases, the total reference counts get increased instead. We examined the assembly codes and found a couple of reasons:

- The native compilers can perform scalar replacement for references to noncontracted arrays. The fused loop body may prevent such scalar replacement due to increased register pressure.
- Loop peeling may decrease the effectiveness of scalar replacement since fewer loop iterations benefit from it.

Execution Time with $ITMAX = 1$

In order to find out whether array contraction increases more data reuses across the time steps (i.e. the T loop iterations), as opposed to within each single time step, we also picked several test programs and measured their performance with $ITMAX$ changed to 1. (Several benchmarks, `hydro2d`, `lucas`, `mg`, `combustion`, `purdue-07`, `purdue-12` and `climate`, already have an artificial outer loop with $ITMAX = 1$. The new experiments are skipped for those programs. We also skipped those programs whose running time with $ITMAX = 1$ is too short to measure by the UNIX `time` utility.)

The results for the remaining five test programs are in Figure 13, which shows that the speedup of the codes after array contraction over the fused codes in Figure 8 is greater than

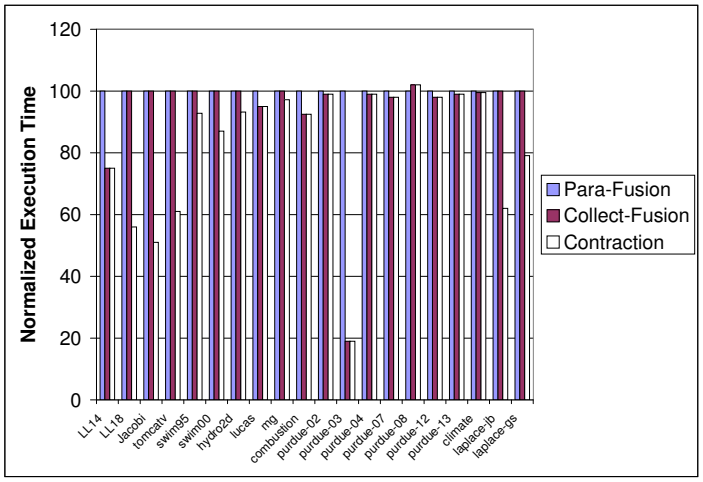


Figure 14: Performance comparision with other schemes

that in Figure 13. This is because parts of program other than the loop nests involved in controlled SFC will consume a relative large portion of total execution time for a smaller *ITMAX*. This results in a smaller speedup.

D.4 Comparison with Other Schemes

Figure 14 compares our contraction technique with other techniques, where "Para-Fusion" stands for parameterized loop fusion [31] and "Collect-Fusion" for collective loop fusion [12]. The execution time for parameterized loop fusion is normalized to 100. The geometric average speedup of SFC over parameterized loop fusion is 1.28 over 20 benchmarks. The average speedup of SFC over collective loop fusion is 1.15.