Fusion of Loops for Parallelism and Locality

Naraig Manjikian and Tarek S. Abdelrahman

Department of Electrical and Computer Engineering
University of Toronto
Toronto, Ontario, Canada M5S 3G4

Abstract

Loop fusion improves data locality and reduces synchronization in data-parallel applications. However, loop fusion is not always legal. Even when legal, fusion may introduce loop-carried dependences which prevent parallelism. In addition, performance losses result from cache conflicts in fused loops. In this paper, we present new techniques to: (1) allow fusion of loop nests in the presence of fusion-preventing dependences, (2) maintain parallelism and allow the parallel execution of fused loops with minimal synchronization, and (3) eliminate cache conflicts in fused loops. We describe algorithms for implementing these techniques in compilers. The techniques are evaluated on a 56-processor KSR2 multiprocessor and on a 16-processor Convex SPP-1000 multiprocessor. The results demonstrate performance improvements for both kernels and complete applications. The results also indicate that careful evaluation of the profitability of fusion is necessary as more processors are used.

Index Terms — Locality enhancement, loop fusion, cache conflicts, loop transformations, data-parallel applications, scalable shared-memory multiprocessors.

1 Introduction

In recent years, scalable shared-memory multiprocessors (SSMMs) have emerged as viable platforms for supercomputing. Several production systems of this type have been introduced, such as the Convex SPP-1000 [9], the Kendall Square Research KSR1/2 [15], and the Cray T3D [10]. Numerous research systems also exist, including the Stanford DASH [19] and FLASH [14], the MIT Alewife [1], and the University of Toronto Hector [29] and NUMAchine [28]. In such systems, the memory is physically distributed to provide scalability, but all processors share a common global address space across the entire memory, as shown in Figure 1. High-speed caches lower the effective access latency for both local and remote memory, and reduce contention. However, the capacity of the caches is limited. As a result, application performance depends not only on how much parallelism is exploited, but also on the degree of locality in the cache.

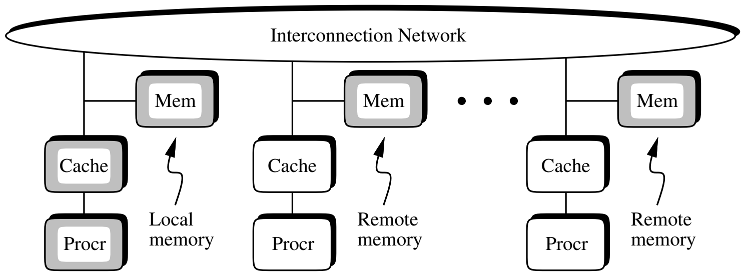


Figure 1: Scalable shared-memory multiprocessor architecture

$$
\begin{array}{ll}
L_a: & \textbf{doall}\ i_1 = \ldots \\
& \quad \ddots \\
& \quad \textbf{doall}\ i_k = \ldots \\
& \quad\quad \ddots \\
& \quad\quad \text{do}\ i_n = \ldots \\
& \quad\quad\quad A_1[F_1^a(\vec{\imath})], A_2[F_2^a(\vec{\imath})], \ldots \\
L_b: & \textbf{doall}\ i_1 = \ldots \\
& \quad \ddots \\
& \quad \textbf{doall}\ i_k = \ldots \\
& \quad\quad \ddots \\
& \quad\quad \text{do}\ i_n = \ldots \\
& \quad\quad\quad A_1[F_1^b(\vec{\imath})], A_2[F_2^b(\vec{\imath})], \ldots \\
& \quad \vdots
\end{array}
$$

Figure 2: Program model

Parallel scientific applications typically consist of sequences of nested loops in which at least $k \geq 1$ outer loops are parallel, as shown in Figure 2. Often, these parallel loop sequences are embedded within a sequential outer loop; we address the presence of sequential outer loops elsewhere [21]. In this paper, we focus on the parallel loop sequence. Iterations from parallel loops may be executed concurrently; synchronization is only required between loop nests to ensure correctness. For efficient parallel execution on SSMMs with caches and a physically-distributed memory, data reuse must be exploited to enhance locality. Reuse can be carried by one or more loops within a single loop nest when the same array element is used in different iterations, or can exist between loop nests when the same array element is used in different loop nests. Exploiting both types of reuse for cache locality has a significant impact on performance.

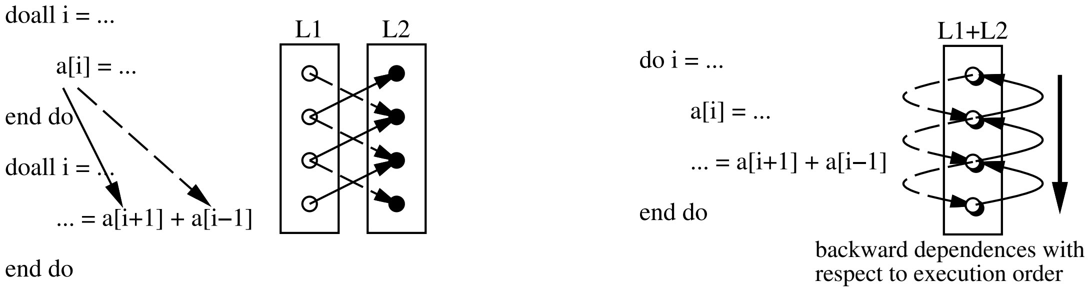
Enhancing locality for individual loop nests has been studied extensively in recent years; example techniques include tiling or blocking [17], and unroll-and-jam [3]. However, techniques for enhancing locality between loop nests have received considerably less attention. This paper presents a new technique to exploit inter-loop data reuse. The technique uses *loop fusion* [3, 6] to bring references in different loop nests into a single loop nest in order to enhance cache locality. However, data reuse between loop nests implies the existence of data dependences, and such dependences may either prevent fusion or force the resulting fused loop nest to be executed serially. In addition, the resulting fused loop nest may reference a large number of arrays, which leads to an increase in the potential for cache conflicts and diminishes the locality benefit of fusion. We specifically propose two novel techniques to overcome the problems arising from fusion. We present a code transformation called *shift-and-peel* which enables loop fusion for locality enhancement by overcoming both fusion-preventing dependences and serializing dependences. This technique maximizes performance by enhancing locality *without* sacrificing the original parallelism in the loops prior to fusion. We couple the shift-and-peel transformation with a data transformation called *cache partitioning* which adjusts the array data layout in memory to prevent the occurrence of cache conflicts which can diminish the locality benefit of fusion.

The remainder of this paper is organized as follows. Section 2 provides background for the problems addressed by our techniques, and reviews related work. Section 3 discusses the shift-and-peel transformation. Section 4 describes cache partitioning to avoid conflicts. Section 5 presents empirical evidence demonstrating the utility and effectiveness of our techniques. Finally, Section 6 provides concluding remarks.

2 Background

2.1 Data Dependence in Nested Loops

The presentation of our proposed techniques requires familiarity with the concepts of data dependence [3, 5, 6, 23]. For two statements S_1 and S_2 which reference a common array a in a nested loop L, a data dependence exists if S_1 references the same array element in some iteration $\vec{i}$ as S_2 in some iteration $\vec{i'}$, where iteration $\vec{i}$ is executed before $\vec{i'}$, and is denoted by $S_1(\vec{i})\delta S_2(\vec{i'})$. The dependence is classified as a *flow* dependence when S_1 writes a and S_2 reads a; as an *anti*dependence when S_1



(a) Loop nests to be fused (b) Illegal fusion of loop nests

Figure 3: Example to illustrate fusion-preventing dependences

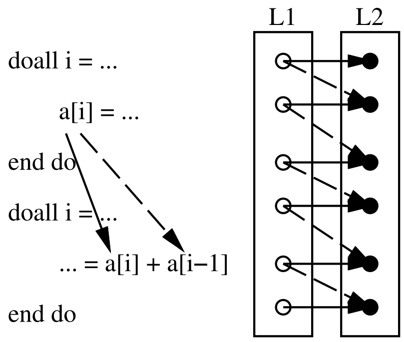
reads and S_2 writes; or as an *output* dependence when both S_1 and S_2 write. For each dependence $S_1(\vec{i})\delta S_2(\vec{i'})$, the dependence distance vector is defined as $\vec{d} = \vec{i'} - \vec{i}$. The dependence direction vector is defined as $\vec{s} = \mathbf{sig}(\vec{d})$. When the distance vector $\vec{d}$ is the same for all dependences between S_1 and S_2, the dependence is said to be *uniform*.

A number of algorithms have been developed to test for dependences. Some tests, such as the Banerjee test [5], can only prove independence, and assume a dependence when independence cannot be proven. These tests do not provide distance information. More recently, algorithms such the Omega test [27] have been developed, which are able to prove dependence, and also provide accurate distance information when a dependence exists. We use such tests since our proposed techniques require dependence distance information.

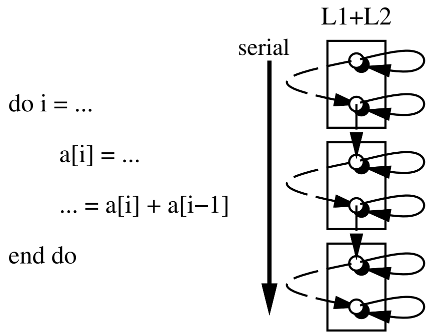
2.2 Loop Fusion

Loop fusion [3, 6] is a code transformation which can be used to combine multiple parallel loop nests into a single loop nest. Fusion can enhance locality by reducing the time between uses of the same data, thereby increasing the likelihood of the data being retained in the cache. Fusion also eliminates the synchronization between parallel loop nests. Fusion is an important transformation because real applications consist of sequences of loop nests which reuse data. However, the reuse implies the existence of data dependences between loop nests, and fusion may violate these dependences. Even when fusion is legal, dependences between the original parallel loop nests may become loop-carried after fusion, serializing the resulting fused loop nest. In the remainder of this section, these problems are illustrated.

The dependences implied by data reuse between loop nests are not loop-carried since their source



(a) Loop nests to be fused



(b) Legal fusion with serializing dependences

Figure 4: Example to illustrate serializing dependences

and sink iterations are in different iteration spaces. Fusion moves the source and sink iterations of each dependence into the same iteration space. If the source and sink are distinct iterations of the fused space, the dependence becomes loop-carried. Fusion is legal only if no resulting loop-carried dependence flows backwards with respect to the iteration execution order [33]. For example, both loops in Figure 3(a) reference the array a. This reuse implies dependences between the iteration spaces, as shown graphically in Figure 3(a). Fusion of the two loops combines the iteration spaces as shown in Figure 3(b), resulting in loop-carried dependences. This fusion is *not* legal because half of the loop-carried dependences flow backwards with respect to the iteration execution order, violating the semantics of the original code. In Figure 3(b), the sink iteration of each dependence would be executed before the source iteration. Consequently, these dependences are referred to as *fusion-preventing* dependences. The remaining dependences are also loop-carried, but flow forwards with respect to the execution order and do not prevent fusion on their own.

Even when there are no fusion-preventing dependences, there may be other dependences prevent parallel execution. This is illustrated in Figure 4. The two loop nests in Figure 4(a) have no loop-carried dependences; the iterations within each loop may be executed in parallel. The only required synchronization is between the loop nests to ensure that all iterations of the first loop nest have been executed before any iterations of the second loop nest are executed. However, when the iteration spaces of the two loop nests are fused as shown in Figure 4(b), loop-carried dependences result. When blocks of iterations from the fused loop nest are assigned to different processors, the synchronization required by these dependences serializes the execution of blocks of iterations. Consequently, these dependences are referred to as *serializing* dependences.

```
doall i = ...
      ... = a[i+1] + a[i−1]
end do

doall i = ...
      ... = b[i+1] + b[i−1]
end do
```

(a) Loop nests prior to fusion

```
doall  i = ...
      ... = a[i+1] + a[i−1]

      ... = b[i+1] + b[i−1]
end do
```

(b) Fused loop nest

Figure 5: Loop fusion and the potential for cache conflicts

2.3 Cache Conflicts

When many loop nests are fused together with the goal of enhancing cache locality, a new problem may arise which leads to a loss of locality. If a large number of arrays is accessed in the body of the fused loop nest, the likelihood of conflicts in the cache increases, even when there is sufficient cache capacity. The occurrence of conflicts is sensitive to cache hardware parameters, array sizes, and data access patterns in the loop nest. These factors make it difficult to guarantee that applying a given loop transformation such as loop fusion for enhancing locality will always improve performance.

The impact of cache conflicts after fusion is more pronounced when exploiting reuse carried across multiple loop iterations for many arrays. This is shown using the example in Figure 5(a), where each loop nest requires retaining data from different arrays in the cache across multiple loop iterations. When the loop nests are fused as in Figure 5(b), data from both arrays must now be maintained simultaneously in the cache across multiple iterations. Because cache capacity is limited, the likelihood of conflicts between the arrays increases, which causes reusable data to be ejected from the cache. As a result, locality is diminished, which is particularly undesirable since the goal of fusion is to enhance locality.

2.4 Related Work

Our techniques stem from previous research in the area of enhancing locality and parallelism in nested loops. Much of this work has focused on individual loop nests. We focus on enhancing locality and parallelism across adjacent loop nests. Previous techniques are limited by dependence constraints, or introduce significant overhead to overcome them. Our contribution with the shift-and-peel transformation is to enhance locality and parallelism across multiple loop nests in spite of dependence constraints, while avoiding the overhead introduced in previous techniques. Although aspects of the shift-and-peel transformation are related to elementary loop transformations such as fusion, strip-mining, peeling, and alignment, our novel formulation to address the limitations of previous techniques is unique. Furthermore, previous research has largely ignored the impact of cache conflicts on locality enhancement. We introduce cache partitioning as a systematic means of ensure that conflicts do not diminish the potential benefits of fusion.

Wolf [31] describes algorithms to improve locality and parallelism in individual nested loops using elementary transformations such as loop permutation for spatial locality and coarse-grain parallelism, and tiling for temporal locality. This work does not address improving temporal locality across sequences of multiple parallel loop nests, thereby missing opportunities for improving performance. In contrast, the shift-and-peel transformation enhances locality with loop fusion while ensuring the transformed code preserves the parallelism of the individual loop nests.

Warren [30] presents an algorithm for incrementally adding candidate loops to a fusible set to enhance vector register reuse, and to permit contraction of temporary arrays into scalars. However, fusion is not permitted in the presence of loop-carried dependences or incompatible loop bounds. In contrast, the shift-and-peel transformation overcomes loop-carried dependences to enable fusion for locality and permit parallel execution, and also addresses differing loop bounds.

Kennedy and McKinley [16] use loop fusion and distribution to enhance locality and maximize parallelism. They focus on enhancing register locality with fusion, and describe a fusion algorithm which prevents fusion of parallel loops with serial loops. However, they disallow fusion when loop-carried dependences result or when the iteration spaces of candidate loops are not identical. In contrast, the shift-and-peel transformation overcomes loop-carried dependences to enable fusion and parallel execution, and also permits loops with differing iteration spaces to be fused.

Porterfield [25] suggests a "peel-and-jam" transformation in which iterations are peeled from the beginning or end of one loop nest to allow fusion with another loop nest. However, no systematic method is described for fusion of multiple loop nests, nor is the parallelization of the fused loop nest considered. In contrast, both the derivation and application of our shift-and-peel transformation from the dependence relationships in a loop nest sequences are systematic, and both locality and parallelism are addressed simultaneously.

Ganesh [11] suggests an extension of Porterfield's peel-and-jam transformation to the inner loops for a pair of multidimensional loop nests. However, as with Porterfield's work, dependences which prevent parallelization are not addressed, nor is a systematic method described.

Callahan [8] proposes loop alignment to allow synchronization-free parallel execution of a loop nest, and uses code replication to resolve conflicts in alignment requirements. This technique may result in an exponential growth in the size of loop bodies to address such alignment conflicts, and the execution of replicated code is a significant source of overhead. In our shift-and-peel transformation, the shifting of iteration spaces used to enable legal fusion is similar to the alignment used to enable parallel execution. However, the shift-and-peel transformation is unaffected by alignment conflicts and enables parallel execution without resorting to replication.

Appelbe and Smith [2] present a graph-based algorithm for deriving the required alignment, replication, and statement reordering to permit parallelization of an individual loop nest with loop-carried dependences. In their derivation, alignment, replication, and reordering are treated as separate cases. However, as with the work of Callahan, this approach incurs overhead due to redundant execution of replicated code. Instead, the shift-and-peel transformation enables parallel execution without the overhead of replication and relies on a simpler graph algorithm to derive the parameters for the transformation.

Pugh [26] presents a method which derives affine schedules for individual statements within a loop to guide transformations for parallelization. Since the aim of this method is optimizing for parallelism, the generated schedules do not permit fusion for locality if loop-carried dependences result. In contrast, our shift-and-peel transformation addresses both locality and parallelism by enabling fusion even in the presence of loop-carried dependences.

Passos and Sha [24] present a multidimensional retiming technique that seeks to remove loop-independent dependences in the body of a single loop nest by moving statement instances between loop iterations. However, this retiming transforms loop-independent dependences into loop-carried dependences, hence the loops must be executed *serially*. As a result, this approach is not appropriate for multiprocessors which exploit parallelism among loop iterations, rather than the limited parallelism within a single instance of the loop body. In contrast, the shift-and-peel transformation enables legal fusion of multiple *parallel* loop nests and enables subsequent parallelization of the fused loops for multiprocessors.

Array padding [3] increases the size of the array dimension aligned with the storage order, and is most effective in reducing the occurrence of self-conflicts in a loop nest when the original array dimensions are powers of 2. Padding is often used in an *ad hoc* manner. The amount of padding which minimizes the occurrence of conflicts between arrays is difficult to determine, especially when data from different arrays must remain cached for reuse. In contrast, cache partitioning systematically determines an array layout in memory to ensure that data from different arrays does not conflict in the cache.

Bacon et al. [4] discuss a method to determine the amount of padding needed to avoid cache and TLB mapping conflicts among individual array references in the *innermost* loop of a loop nest. Their method is heuristic, involving a search for an appropriate value of padding for each array. Their approach does not consider data reuse in outer loops, which is particularly important when applying loop fusion. As such, it cannot necessarily prevent cache conflicts for all reusable data. In contrast, cache partitioning prevents conflicts for data reuse carried by outer loops.

Memory alignment [3] is another data-reorganization technique which aligns data in memory to cache line boundaries in a effort to contain individual data items within a single cache line whenever possible. While this can reduce cache traffic and decrease the potential for conflicts for small data sets, the impact is not significant for large data sets, such as arrays in scientific loop nests, because the cache line size is relatively small.

Lebeck and Wood [18] present a case study of improving performance with a variety of techniques including data transformations such as padding and memory alignment. However, these transformations are discussed in the context of programmer tuning of application performance. There is no discussion of how to incorporate such transformations in a compiler.

In summary, the shift-and-peel transformation presented in this paper enables fusion and subsequent parallelization of multiple loop nests even in the presence of fusion-preventing or serializing dependences. In particular, parallelization is enabled without the costly replication in the techniques proposed by Callahan [8], and Appelbe and Smith [2]. This paper also describes cache partitioning to address the potential for cache conflicts that result from bringing references to many different arrays closer together with fusion, particularly when reuse is carried by outer loops in the fused loop nest. This is an aspect of the fusion problem which has not been addressed in the past.

3 The Shift-and-Peel Transformation

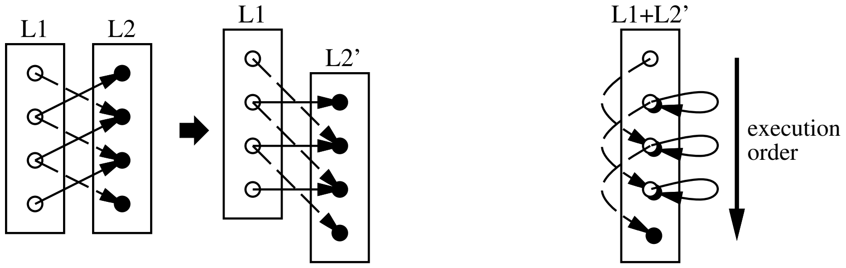
In this section, the shift-and-peel transformation is described. The basic idea of the technique is described first, followed by a description of the algorithms for applying the transformation on sequences of multidimensional parallel loop nests.

3.1 Shifting to Enable Legal Fusion

We propose a technique to enable legal fusion in the presence of backward loop-carried dependences. This technique only requires uniform dependences. The basic idea is to make backward dependences loop-independent in the fused loop by *shifting* the iteration space containing the sink of the dependences with respect to the iteration space containing the source of the dependence, which is similar to the alignment techniques described in [8, 25]. The amount by which to shift is determined by the dependence distance. Other dependences between the loops may be affected, but do not prevent fusion. After this shifting, the loops can be legally fused. We illustrate this procedure in Figure 6, using the iteration spaces shown earlier in Figure 3. The iteration space of the second loop in Figure 6(a) must be shifted by one iteration because of the backward dependence with a distance of one. The shift increases the distance of the forward dependence. The loops may then be legally fused, as shown in Figure 6(b). The algorithm for deriving the required amount of shifting for arbitrary sequences of loop nests is discussed in Section 3.3.

3.2 Peeling to Enable Parallel Execution of Fused Loop Nests

We propose a technique to remove serializing dependences resulting from fusion of parallel loops. The technique requires uniform dependences between the loop nests, and static, blocked scheduling of the fused loop. Static scheduling is not a serious limitation, as it is normally the most efficient approach when the computation is regular with uniform dependences. The basic idea is to identify



(a) Shifting of iteration spaces to make backward dependences loop–independent

(b) Legal fusion after shifting iteration spaces

Figure 6: Shifting iteration spaces to permit legal fusion

iterations from the *original* loop nests which become sinks of cross-processor dependences[1] in the fused loop, then *peel* these iterations from their respective iteration spaces prior to fusion. In this manner, cross-processor dependences are removed between blocks of iterations from the resulting fused loop which are assigned to different processors. The peeled iterations may be executed legally after the fused loop has completed. Since the dependences are uniform and block scheduling is used, the peeled iterations are located at block boundaries. The number of iterations which must be peeled is determined by the dependence distance. This procedure is illustrated in Figure 7 using the iteration spaces shown previously in Figure 4. Since the distance of the forward loop-carried dependence is one, one iteration is peeled from the iteration space of the second loop at each of the block boundaries. When the two loops are fused in Figure 7, the blocks of iterations may be executed in parallel on different processors. Loop-carried dependences still exist, but are contained entirely within a block of iterations executed by the same processor. The peeled iterations are executed only after all of the blocks have been executed in parallel. The peeled iterations may themselves be executed in parallel. The algorithm for deriving the required amounts of peeling for arbitrary sequences of parallel loop nests is given in Section 3.3.

3.3 Derivation of Shift-and-Peel

In general, two or more multidimensional loop nests may be considered for fusion, and fusion-preventing or serializing dependences may result from any pair of loop nests in the candidate set. Complex dependence relationships may exist between candidate loop nests in the form of *dependence chains* passing through iterations in different loop nests. These dependence chains are

[1]Cross-processor dependences are loop-carried dependences for which the source and sink iterations are executed by different processors.

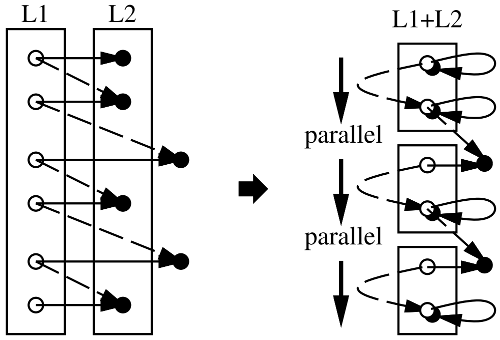


Figure 7: Peeling to retain parallelism when fusing parallel loops

dictated by the array reuse and constitute iteration ordering constraints which must be satisfied for correctness. If shifting or peeling is applied to one loop nest, all subsequent loop nests along all dependence chains which pass through the affected loop nest must also be shifted or peeled in order to satisfy the ordering constraints for the affected iterations. That is, shifting and peeling must be *propagated along dependence chains.* It is advantageous to treat candidate loop nests for fusion collectively rather than incrementally one pair at a time.

We present algorithms to determine the amount of shifting and peeling needed for each iteration space to permit legal fusion of a collection of multidimensional loop nests, and subsequent parallelization of the fused loop nest. We require that dependences between iterations in different loop nests are uniform in those dimensions of the iteration spaces which are being fused. The technique is applied to each dimension, working inward from the outermost loop level. An acyclic *dependence chain multigraph* is constructed for each dimension to represent the dependence chains. Because the dependences are uniform, so are the dependence chains. Each loop nest is represented by a vertex, and each dependence between a pair of loop nests in that dimension is represented by an edge weighted by the corresponding dependence distance. Fusion collapses multiple iteration spaces into one, hence all statements in a fused loop will share the same loop index variable. This fact can be exploited by assuming that the index variables of the different loops are the same [33]. The dependence distances between iterations from different loops can be easily determined from the subscript expressions for each array reference. A forward dependence has a positive distance, and results in an edge with a positive weight. Conversely, a backward dependence has a negative distance, and results in an edge with a negative weight. A multigraph is required since there may

```
TRAVERSEDEPENDENCECHAINGRAPH(G)::
    foreach v ∈ V[G] do weight(v) = 0 endfor
    foreach v ∈ V[G] in topological order do
        foreach e = (v, v_c) ∈ E[G] do
            if weight(e) < 0 then weight(v_c) = min(weight(v_c), weight(v) + weight(e))
                             else weight(v_c) = min(weight(v_c), weight(v))
            endif
        endfor
    endfor
```

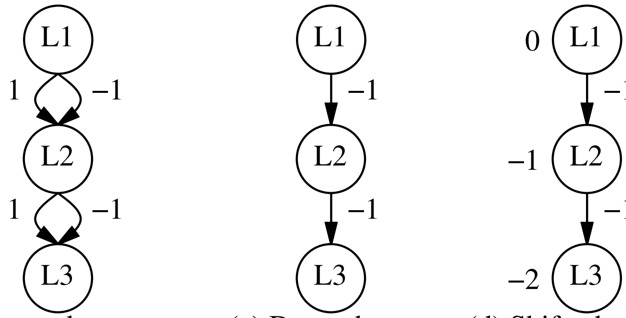
Figure 8: Algorithm for propagating shifts along dependence chains

be multiple dependences between the same two loop nests.

In deriving the required amounts of shifting, the dependences of interest are those with negative distances. The multigraph is reduced to a simpler *dependence chain graph* by replacing multiple edges between two vertices by a single edge whose weight is the *minimum* of the original set of edges between these two vertices. When this minimum is negative, it determines the shift required to remove all backward dependences between the two corresponding loop nests. This reduction preserves the structure of original dependence chains. A traversal algorithm is then used to propagate shifts along the dependence chains in this graph. Each vertex is assigned a weight, which is initialized to zero, and the vertices are visited in topological order to accumulate shifts along the chains. Note that this topological order is given by the original loop nest order, hence there is no need to perform a topological sort. Only edges with a negative weight contribute shifts; all other edges are treated as having a weight of zero and serve only to propagate any accumulated shifting. At each vertex, the minimum value for all accumulated shifts through that vertex is always selected to ensure that all backward dependences are removed. The algorithm is given in Figure 8. The complexity of the algorithm is linear in the size of the graph, and upon termination, the final vertex weights indicate the amount by which to shift each loop *relative to the first loop* to enable legal fusion. Figure 9 illustrates the above procedure for deriving shifts.

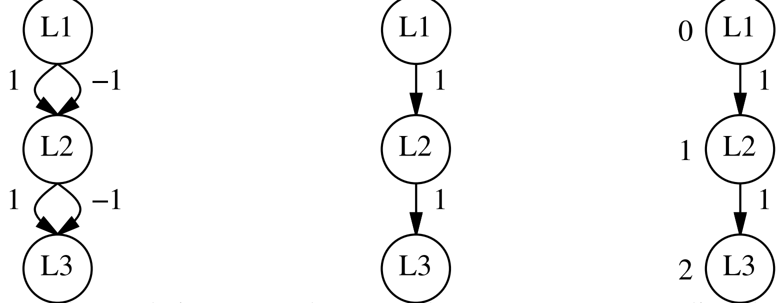
In deriving the required amounts of peeling, the original dependence chain multigraph is reconsidered. This time, the edges of interest are those with positive weights. The multigraph is reduced to a simpler graph by replacing multiple edges between two vertices with a single edge whose weight is the *maximum* from the original set of edges between these two vertices (as opposed to the minimum as in the case of shifting). The reduced graph preserves the dependence chains

```
L1: do i=2,n−1
      a[i] = b[i]
    end do
L2: do i=2,n−1
      c[i] = a[i+1]+a[i−1]
    end do
L3: do i=2,n−1
      d[i] = c[i+1]+c[i−1]
    end do
```



(a) Example loop nest sequence (b) Dependence chain multigraph (c) Dependence chain graph (d) Shifts derived from traversal

Figure 9: Representing dependences to derive shifts for fusion



(a) Dependence chain multigraph (b) Dependence chain graph (c) Peeling derived from traversal

Figure 10: Deriving the required amount of peeling

from the original multigraph and remains acyclic. The graph traversal algorithm is used again to propagate the required amounts of peeling along the dependence chains. The only modification is to consider edges with a positive weight, since only they require peeling to remove cross-processor dependences; all other edges are treated as having a weight of zero to propagate any accumulated amounts of peeling. Upon termination, the final vertex weights are the number of iterations to peel relative to the first loop. Figure 10 illustrates this procedure using the dependence chain multigraph shown in Figure 9(a).

3.4 Implementation of Shift-and-Peel

Once the required amounts of shifting and peeling have been derived, the loop nests must be transformed appropriately to complete the legal fusion. There are two methods to implement shift-and-peel. In a direct approach, the bodies from the original loop nests are combined into a single body such that the computation performed in related iterations of the original loop nests is all performed in one iteration of the fused loop. The iterations of the fused loop are then divided

```
do i=istart,iend
    a[i] = b[i]
    if (i  >=  istart+1) c[i−1] = a[i]+a[i−2]
    if (i  >=  istart+2) d[i−2] = c[i−1]+c[i−3]
end do

c[iend] = a[iend+1] + a[iend−1]

do i=iend−1,iend
   d[i] = c[i+1]+c[i−1]
end do
```

(a) Direct method

```
do ii=istart,iend,s
   do i=ii,min(ii+s−1,n−1)
      a[i] = b[i]
   end do
   do i=max(ii−1,istart+1),min(ii+s−2,iend−1)
      c[i] = a[i+1]+a[i−1]
   end do
   do i=max(ii−2,istart+2),min(ii+s−3,iend−2)
      d[i] = c[i+1]+c[i−1]
   end do
end do

c[iend] = a[iend+1]+a[iend−1]
do i=iend−1,iend
   d[i] = c[i+1]+c[i−1]
end do
```

(b) Strip−mined method

Figure 11: Alternatives for implementing fusion with shift-and-peel

into blocks to be executed in parallel by each processor. To implement shifting, array subscript expressions in statements from shifted loop nests must be adjusted wherever the index variable of the shifted loop appears. To implementing peeling, guards must be introduced for each statement from a loop which requires peeling. Figure 11(a) illustrates this approach for a block of iterations `istart...iend` executed by one processor. Because of shifting, some iterations from shifted loops are executed outside the fused loop.

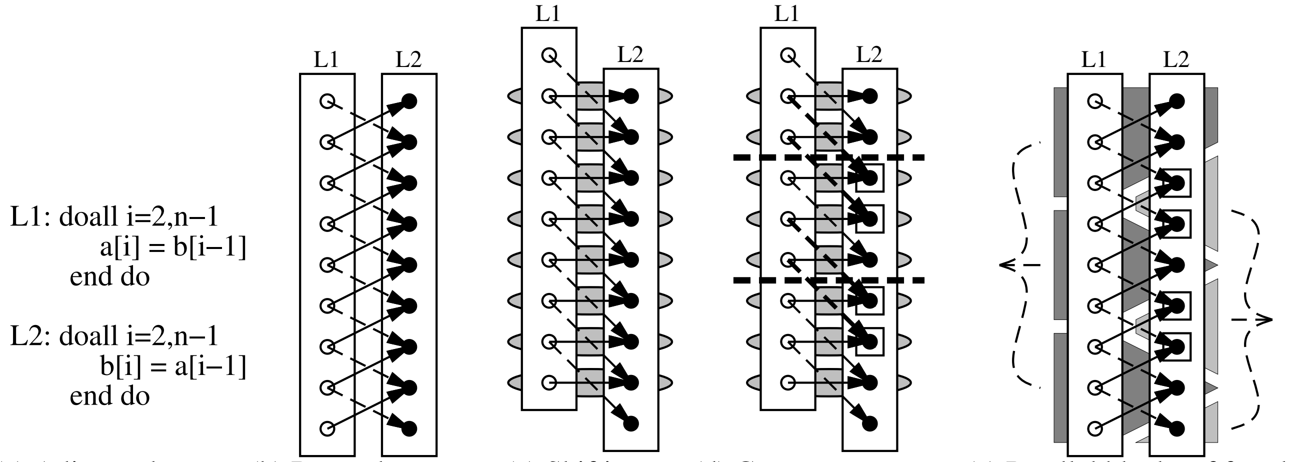
An alternative to the direct approach is to strip-mine [3, 6] the original loops by a factor of s, then fuse the resulting outer controlling loops, as shown in Figure 11(b). Implementing shifts for this method only requires adjusting the inner loop bound expressions with the amount of the shift, leaving the subscript expressions unchanged. Implementing peeling also requires an adjustment to the inner loop bound expressions. The strip-mined method offers several advantages over the direct method: (a) array subscript expressions are unchanged, (b) register pressure is decreased, and (c) strip-mining accommodates differing iteration spaces by suitable modifications to the inner loop bounds. Strip-mining may incur some overhead in comparison to the direct approach, but the strip size may be used to determine the amount of data loaded into the cache for each array referenced in the inner loops. This flexibility is desirable in controlling the extent of cache conflicts, as will be described in Section 4. In light of these advantages, we elect to use strip-mining to implement fusion with shift-and-peel.

```
do ii=istart,iend,s
   do i=ii,min(ii+s−1,iend)
      a[i] = b[i]
   end do
   do i=max(ii−1,istart+1),min(ii+s−2,iend−1)
      c[i] = a[i+1]+a[i−1]
   end do
   do i=max(ii−2,istart+2),min(ii+s−3,iend−2)
      d[i] = c[i+1]+c[i−1]
   end do
end do
<BARRIER>
do i=iend,iend+1
   c[i] = a[i+1]+a[i−1]
end do
do i=iend−1,iend+2
   d[i] = c[i+1]+c[i−1]
end do
```

Figure 12: Complete implementation of fusion with shift-and-peel

The only remaining issue is the execution of the iterations peeled to enable parallel execution. These iterations are peeled from the start of each block on different processors and can only be executed after all preceding iterations have been executed; a barrier synchronization can be inserted to ensure that this condition is satisfied. Iterations peeled from the same block are grouped into sets. There are no dependences between different sets of peeled iterations, although there may be dependences within each set. As a result, these sets of peeled iterations may also be executed in parallel without synchronization.

Shifting results in a number of iterations which are executed outside the fused loop. These iterations are at the end of blocks assigned to different processors. Because there may be dependences between the iterations at the end of a block assigned to one processor, and the iterations peeled from the start of the adjacent block assigned to another processor, we collect such iterations from adjacent processors into sets such that all dependences are contained entirely within each set. In this manner, these sets of iterations may be executed in parallel. Figure 12 illustrates the complete code which implements fusion with shift-and-peel. Iterations peeled to enable parallel execution are executed after a barrier to ensure all preceding iterations have been executed. The iterations executed after the barrier include those from shifted loops at the end of block `istart...iend`, and also those peeled from the start of the block beginning at `iend+1`.



(a) Adjacent loops to be fused (b) Dependences between loops (c) Shifting for fusion (d) Cross−processor dependences (e) Parallel blocks of fused and peeled iterations

Figure 13: Legality of the shift-and-peel transformation

3.5 Legality of the Shift-and-peel Transformation

In this section, we argue intuitively that the shift-and-peel transformation is always legal for a sequence of parallel loops with uniform dependences between iterations in different loops. A formal proof of legality is provided in Appendix I.

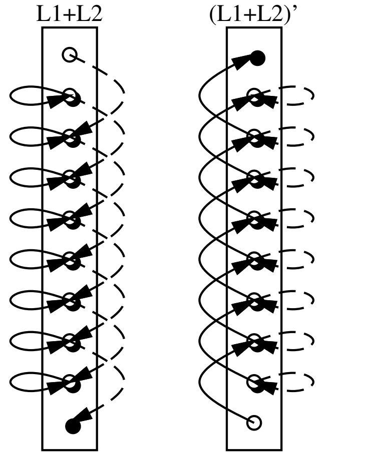
Consider the example sequence of parallel loops in Figure 13(a). This example is representative of the loop nests we consider because it contains both forward and backward dependences between the two loops. These dependences are illustrated in Figure 13(b). Because each loop is parallel, there are no loop-carried dependences within the individual loops. The antidependence between L_1 and L_2 caused by references to the array b is uniform with a distance of -1, and hence it prevents fusion. In general, there may be several such fusion-preventing dependences with different distances. The derivation algorithm in Figure 8 always selects the amount of shifting according to the minimum dependence distance between the loops. Similarly, the flow dependence for array a is also uniform with a distance of 1, hence it serializes execution if the backward dependence is ignored and the loops are fused. In general, there may be several serializing dependences with different distances. In our derivation algorithm, the amount of peeling is always determined by the dependence with the maximum distance, as discussed in Section 3.3.

Based on the antidependence with the distance of -1, L_2 is shifted by one iteration with respect to L_1 to permit legal fusion. This is shown in Figure 13(c). The computation performed in each pair of iterations identified by the shading in Figure 13(c) is performed in one iteration of the resulting fused loop. The original dependence distance of -1 is transformed to 0, since it is the minimum distance. All other dependence distances, including the forward dependence distance of 1, are increased, but this does not prevent legal fusion. Hence it is always legal to perform fusion after shifting by the amount needed to satisfy the minimum dependence distance.

Now, consider a parallel execution of the fused loop. This is shown in Figure 13(d), where each processor is assigned a contiguous block of iterations. There are now cross-processor dependences which flow from iterations in one processor to iterations in another processor, and hence the blocks of iterations must be executed serially. The iterations from L_2 identified with a square in Figure 13(d) are the sink iterations of these cross-processor dependences. In the absence of shifting, some of these sink iterations would otherwise be executed in the same processor as their corresponding source iterations. However, shifting moves each of these sink iterations to an adjacent processor. The number of such iterations per block is equal to the amount of shifting. The remaining sink iterations would still generate cross-processor dependences even without shifting because of the forward dependence between the original loops. The number of such iterations is equal to the maximum distance among all original forward dependences, and this number is determined by applying the derivation algorithm in Figure 8 to determine the amount of peeling. For the example in Figure 13, there is one such iteration per block.

To remove all serializing constraints and permit parallel execution, the iterations which would otherwise become sinks of cross-processor dependences are peeled out of L_2 prior to performing fusion. Of each pair of iterations peeled from the blocks of iterations in Figure 13(d), one must be peeled out as a consequence of shifting, and the other due to the original forward dependence with a distance of 1. The shift-and-peel transformation thus groups the computations into the blocks of fused and peeled iterations shown in Figure 13(d). The blocks of fused iterations are executed in parallel, then the blocks of peeled iterations are executed in parallel after a barrier synchronization.

Using Figure 13, we argue that the shift-and-peel transformation is always legal. First, no dependences flow between blocks of fused iterations by virtue of peeling all iterations that would

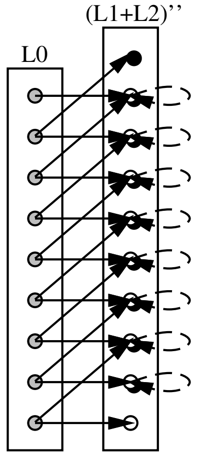


(a) Alignment conflicts

```
L0:    doall i=1,n−2
          b0[i]=b[i]
       end do

(L1+L2)'': doall i=1,n−1
             if  (i>1)  a[i] = b0[i−1]
             if  (i<n−1) b[i+1]=a[i]
           end do
```

(b) Array replication



(c) Parallel loop after replication

Figure 14: Resolution of alignment conflicts with replication

otherwise serialize execution. Within each block of fused iterations, shifting to satisfy the minimum dependence distance ensures that the fusion is indeed legal, as shown in Figure 13(c). Second, no dependences flow between blocks of peeled iterations. Dependences either flow from a block of fused iterations to a block of peeled iterations, or they flow within the same block of peeled iterations, and are satisfied by the order in which they are executed within the block of peeled iterations. Finally, since dependences only flow from blocks of fused iterations to blocks of peeled iterations, the barrier synchronization ensures that these are always satisfied.

It is interesting to note that dependence relationships in the fused loop shown in Figure 13(c) lead to an alignment conflict which requires replication if parallel execution is enabled using the techniques proposed by Callahan[8] and Appelbe and Smith [2]. This conflict is illustrated in Figure 14(a). The loop which results from fusion is serial due to a forward loop-carried dependence. This forward dependence is the flow dependence for array a. If the computations in the loop are aligned as shown in Figure 14(a) such that the forward dependence is made loop-independent, a backward dependence results, hence the alignment is illegal. Alignment for parallel execution is not possible because the requirements of the different dependences conflict with each other. To resolve this alignment conflict, replication is required. In Figure 14(b), a new loop L_0 replicates the array b into a new array $b0$, and the values of array $b0$ are read in the aligned version of the fused loop, rather than array b. As a result, the backward loop-carried dependence is removed, and the

```
do j=2,n-1
  do i=2,n-1
    b[i,j] = (a[i,j-1]+a[i,j+1]
             +a[i-1,j]+a[i+1,j])/4
  end do
end do

do j=2,n-1
  do i=2,n-1
    a[i,j] = b[i,j]
  end do
end do
```

(a) Original loop nest sequence

```
do jj=2,n-1,sj
  do ii=2,n-1,si
    do j=jj,min(jj+sj-1,n-1)
      do i=ii,min(ii+si-1,n-1)
        b[i,j] = (a[i,j-1]+a[i,j+1]
                 +a[i-1,j]+a[i+1,j])/4
      end do
    end do
    do j=max(jj-1,2),min(jj+sj-2,n-2)
      do i=max(ii-1,2),min(ii+si-2,n-2)
        a[i,j] = b[i,j]
      end do
    end do
  end do
end do

do j=2,n-2
  do i=n-1,n-1
    a[i,j] = b[i,j]
  end do
end do
do j=n-1,n-1
  do i=2,n-1
    a[i,j] = b[i,j]
  end do
end do
```

(b) Fused loop nest sequence with peeled iterations due to shifting

Figure 15: Fusion with multi-dimensional shifting for Jacobi loop nest sequence

aligned loop is not only legal, but may also be executed in parallel, since it no longer contains any loop-carried dependences. The new loop L_0 may also be executed in parallel. However, L_0 may not be fused with the aligned loop because the original alignment conflict would then reappear.

In general, both data *and* computation may require replication to address alignment conflicts within a loop and permit parallel execution. Replicating computation contributes execution time overhead, while replicating data contributes memory overhead. In contrast, our shift-and-peel transformation avoids entirely the overhead of replication in the fused loop, exploiting reuse for locality while preserving the parallelism found in the original loop sequence.

3.6 Code Generation for Multidimensional Shift-and-Peel

Fusing multi-dimensional loop nests with strip-mining for serial execution does not present difficulties since only shifting is required. The loops being fused are strip-mined, the control loops are pushed to the outermost level, and then the control loops are fused. The appropriate amounts of shifting are reflected directly in the inner loop bounds. Finally, iterations moved out of the fused loop as a result of shifting are executed. More than one loop nest is required for this purpose because these iterations are not contained in a single rectangular region.

This procedure is illustrated using the Jacobi loop nest sequence in Figure 15(a). The dependences in this sequence of two loop nests are such that the second loop nest requires shifting of one iteration and peeling of one iteration in both outer and inner loops. The fused loop nest sequence

```
JNPROCS = <#processors along j−dimension>
INPROCS = <#processors along i−dimension>
jp = mypid / JNPROCS
ip = mypid % INPROCS
jblksz = j_trip_count / JNPROCS
iblksz = i_trip_count / INPROCS
jstart = 2+jp * jblksz
istart = 2+ip * iblksz
if (jp == JNPROCS − 1)
  jend = n−1
else
  jend = jstart + jblksz
endif
if (ip == INPROCS − 1)
  iend = n−1
else
  iend = istart + iblksz
endif
left     =  (ip == 0)
right    =  (ip == INPROCS − 1)
top      =  (jp == 0)
bottom =  (jp == JNPROCS − 1)
jfpeel  =  (left) ? 0 : 1
ifpeel  =  (top) ? 0 : 1
jppeel  =  (right) ? 0 : 1
ippeel  =  (bottom) ? 0 : 1
```

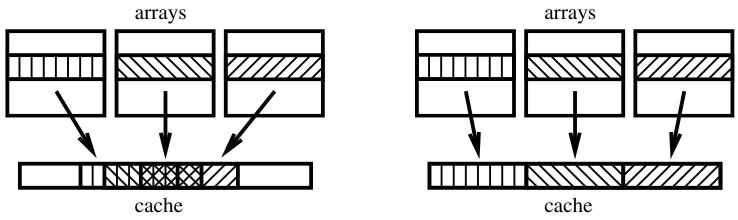
```
do jj=jstart,jend,sj
  do ii=istart,iend,si
    do j=jj,min(jj+sj−1,jend)
      do i=ii,min(ii+si−1,iend)
        b[i,j] = (a[i,j−1]+a[i,j+1]
                  +a[i−1,j]+a[i+1,j])/4
      end do
    end do
    do j=max(jj−1,jstart+jfpeel),min(jj+sj−2,jend−1)
      do i=max(ii−1,istart+ifpeel),min(ii+si−2,iend−1)
        a[i,j] = b[i,j]
      end do
    end do
  end do
end do
<BARRIER>
do j=jstart,jend−1
  do i=iend,iend+ippeel
    a[i,j] = b[i,j]
  end do
end do
do j=jend,jend+jppeel
  do i=istart,iend+ippeel
    a[i,j] = b[i,j]
  end do
end do
```

Figure 16: Parallelization with multi-dimensional peeling for Jacobi loop nest sequence

with shifting only is shown Figure 15(b).

However, peeling to enable parallel execution introduces some minor complications. The complications arise from the fact that the multidimensional iteration space is divided into blocks of iterations which are executed by different processors. For those processors which execute blocks on the boundary of the multidimensional space, there are slight differences in the generated code. When parallelizing a single loop, there are only three cases: the starting boundary block, all interior blocks, and the ending boundary block. This number is small enough to permit generating three different versions of the code. However, when more than one loop in the fused loop nest is parallelized, the number of cases increases dramatically. For example, if two loops are parallelized, there are four corner blocks, four different classes of side boundary blocks, and all interior blocks, resulting in a total of nine cases. With three loops parallelized, the top and bottom faces contribute nine cases each, the four remaining faces and four lateral edges contribute eight more cases, and finally all interior blocks must be considered, giving a total of 27 cases. Clearly, it is not feasible to generate 27 different versions of the code, particularly because the differences are quite minor.

The differences between the various cases center on the execution of the iterations peeled



(a) Conflicting mapping of array data into overlapping cache regions

(b) Conflict−free mapping of array data obtained with cache partitioning

Figure 17: Avoiding conflicts in the cache

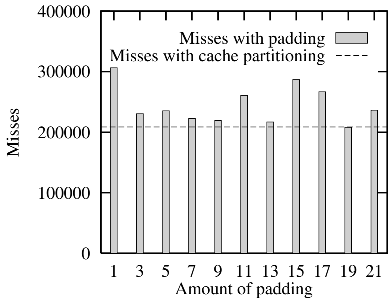


Figure 18: Cache misses for `LL18`

to enable parallel execution. Instead of generating multiple versions, only one set of loops is generated, with the different cases reflected in a number of variables which control peeling in the fused loops and the subsequent execution of peeled iterations. The values for these control variables are determined by a prologue to the fused loop nest that computes which case this instance of the code represents. The prologue determines which boundary or boundaries the block of iterations includes, then sets the flags to control the peeled iterations accordingly.

This approach is illustrated for the Jacobi loop nest sequence in Figure 16. The iteration space is divided into rectangular blocks for distribution on a grid of processors.

4 Cache Partitioning

Conflicts among data items in the cache cause misses that diminish locality. *Self*-conflicts occur between elements from the same array, while *cross*-conflicts occur between elements from different arrays [17]. Cross-conflicts are particularly serious when fusing multiple loop nests which together reference a large number of arrays. These conflicts occur when portions of data from different arrays map into overlapping regions of the cache, as illustrated in Figure 17(a). Reuse of array data may extend across multiple iterations, and shifting to enable legal fusion increases the distance between reuses. The net effect is an increase in the total amount of data which must remain cached in order to enhance locality, and hence an increase in the potential for conflicts. There are a number of hardware techniques such as increasing set-associativity for addressing the potential for cache conflicts, but these are beyond the scope of this paper.

A common software technique is to *pad* inner array dimensions to perturb the mapping of data into the cache and reduce the occurrence of conflicts [3]. Padding is particularly useful in preventing self-conflicts for a given array, especially when array dimensions are powers of 2. However, large cache sizes reduce the likelihood of self-conflict. It is more difficult to predict the amount of padding which minimizes the number of cross-conflicts, particularly when the number of arrays is large. Figure 18 depicts the impact of various amounts of padding on the number of cache misses for the execution of a fused loop referencing nine arrays whose original dimensions are 512×512 (the details of this experiment will be described in Section 5). The number of misses varies erratically with the amount of padding, making it difficult to select the amount of padding to use, and the minimum occurs for a relatively large padding of 19, which leads to considerable amount of unused memory.

We propose *cache partitioning* as a systematic means of avoiding conflicts between references to different arrays without the "guesswork" of padding. Rather than perturb the mapping of data into the cache in an unpredictable manner by padding *within* inner array dimensions, cache partitioning seeks to precisely adjust the array layout in memory with padding *between* arrays. To avoid confusion, we refer to the padding between arrays as *gaps*. The basic idea behind cache partitioning is to logically divide the available cache capacity into non-overlapping partitions, one for each array. This partitioning is done entirely in software; no hardware support is required. The array starting address are then adjusted by inserting appropriately-sized gaps between the arrays such that each array maps into a different partition, as shown in Figure 17(b). Evidence of the effectiveness of cache partitioning can be seen in Figure 18, which compares the number of misses from applying cache partitioning using the *original* array sizes with the number of misses for various amounts of internal padding. Cache partitioning directly minimizes the number of misses and prevents erratic cache behavior.

As indicated in Figure 17, we do not assume that arrays fit entirely within the cache. Rather, we are concerned with the more typical case when arrays do not fit in the cache, which requires locality-enhancing transformations such as fusion of adjacent loop nests. To ensure that the full locality benefit of fusion is realized, cache partitioning permits *portions* of data from different arrays to remain cached without conflicting with each other, and is most effective when the reuse being exploited is associated with uniform dependences [12, 32], and when the data access patterns are compatible.

However, it must be noted that cache partitioning is also very effective when the total data size for all arrays is less than the cache capacity. In this case, cache partitioning ensures that no data is ejected due to conflicts between array references throughout the execution of all loops which access the arrays in question. As a result, locality-enhancing loop transformations would not be necessary. However, it is more reasonable to expect that for typical applications, the data does not fit in the cache, in which locality-enhancing transformations must be applied, and cache conflict avoidance with cache partitioning is needed to ensure that these transformations realize their full benefit.

A pair of references $A(f(\vec{\imath})), A(g(\vec{\imath}))$ to the same array A within a loop nest with iteration vector $\vec{\imath}$ generate uniform dependences if $f(\vec{\imath}) = h_A \cdot \vec{\imath} + c_f$ and $g(\vec{\imath}) = h_A \cdot \vec{\imath} + c_g$. The mapping $h_A : \mathbf{Z}^\ell \mapsto \mathbf{Z}^k$ is represented by a $k \times \ell$ matrix, where k is the array dimensionality and ℓ is the loop nest dimensionality, and c_f, c_g are constant vectors in $\mathbf{Z}^k$. In the presence of uniform dependences, spatial and temporal reuse can be effectively exploited with appropriate transformations. In the absence of uniformity, any potential reuse may be difficult to exploit [32].

Intuitively, references to different arrays in a loop nest are *compatible* if they have the same stride and direction. More formally, the references for a pair of arrays A and B are compatible if $h_A = h_B$. Compatibility ensures that once the mapping of array starting addresses into the cache is made conflict-free with cache partitioning, the mapping for the remaining data will also be conflict-free throughout the execution of the loop nest. Initially, data from each array is loaded into separate portions of the cache determined by the starting addresses. As the execution of the loop nest proceeds, new data from each array is loaded into the cache, overwriting older data whose reuse has been completed. As a result, the partition boundaries move through cache during the execution of the loop nest. However, compatibility ensures that the partitions never overlap so that conflicts between portions of different arrays do not occur.

Compatibility may seem to be a restrictive requirement, but it is normally satisfied in the presence of uniform dependences. Moreover, the impact of conflicts in the presence of compatibility

```
GreedyMemoryLayout(A, c)::                    // A = set of arrays, c = cache size
n_a = |A|
s_p = c/n_a                                   // partition size
P = {0, 1, ..., n_a − 1}                      // available partition indices
q = q_0                                       // starting address of available storage
do
  select a ∈ A                                // selection is arbitrary
  mapped_cache_address = CacheMap(q)
  foreach p ∈ P do                            // determine gaps for available partitions
    target_cache_address(p) = p · s_p
    gap(p) = target_cache_address(p)− mapped_cache_address
    if target_cache_address(p) < mapped_cache_address then
      gap(p) = gap(p) + cache_size            // “wraparound” in the cache
    endif
  endfor
  select p_opt ∈ P where gap(p_opt) = min_{p∈P} gap(p)   // select minimum gap
  P = P \ {p_opt}                             // remove from available partitions
  START(a) = q + gap(p_opt)                   // insert gap
  q = START(a) + SIZE(a)                      // adjust start for next array
  A = A \ {a}
while A ≠ ∅
```

Figure 19: Greedy memory layout algorithm for cache partitioning

is particularly serious, because conflicts occur frequently in regular patterns once they begin. Hence, compatibility is an important case. Where necessary, it is often possible to obtain compatibility from uniform dependences with appropriate data transformations. For example, if h_A and h_B are identical except for a permutation of rows, the array dimensions corresponding to those rows in one of the arrays may be permuted to obtain compatibility. If h_A and h_B differ in the stride for one dimension, array compression or expansion [18] along that dimension can be applied. If h_A and h_B differ in the sign in one dimension, the storage order in that dimension can be reversed for one of the arrays. In conjunction with code transformations (e.g., loop permutation), such data transformations improve the utilization of cache lines, i.e., exploit spatial reuse.

Cache partitioning adjusts array starting addresses by introducing appropriately-sized gaps between arrays in memory. These gaps represent memory overhead which should be minimized. In this paper, we limit our presentation to *one-dimensional* cache partitioning to correspond with fusion of the outermost loop in a sequence of loop nests; higher-dimensional cache partitioning is described in [20]. We employ the greedy layout algorithm shown in Figure 19 to reduce the size of these gaps for a set of n_a arrays which are referenced in a loop nest (e.g., the result of fusion). The algorithm determines the starting addresses of the partitions in the cache, each of size s_p, then performs the array memory layout such that the starting address of each array maps to a different partition of the cache. A set of available partitions P is maintained, and arrays are selected one by one in an arbitrary order to be placed in memory. The greedy nature of the algorithm is reflected in the choice of a partition $p_{opt} \in P$ which minimizes the size of the gap to be inserted between the end of the previously-placed array and the current array. Each partition selected in this manner is removed from the set P to ensure that two arrays are not assigned to the same partition. The algorithm assumes a direct-mapped cache with a typical address mapping function $CacheMap()$. The complexity of the algorithm is $O(n_a^2)$, and the final layout results in a conflict-free mapping for compatible array references.

It is important to note that cache partitioning is also applicable for set-associative caches, although the benefit is reduced because set-associativity alone can reduce the occurrence of conflicts in hardware. However, conflicts may still occur because the number of arrays resulting from fusion of multiple loop nests may exceed the associativity, and the hardware-implemented replacement policy may make inappropriate selections of data awaiting reuse to be ejected from the cache. The algorithm in Figure 19 may be applied with one minor modification for a set-associative cache with associativity a. The size s_p of each partition remains the same, but *target_cache_address*(p) is computed as $\lfloor p/a \rfloor \cdot s_p$ to reflect the associativity of a.

The partition size directly determines the maximum strip-mining size for fusion using approach discussed in Section 3.4, where the largest possible strip size reduces the overhead of strip-mining. The strip size must be such that the total data referenced for each array in the inner loops fits within a cache partition. Selection of a larger strip size is legal, but causes data to overflow into neighboring partitions in the cache, leading to unnecessary conflicts and reducing performance.

The advantage of cache partitioning is that it results in predictable cache behavior by systematically avoiding the conflicts which diminish the benefit of locality, particularly for fusion. The overhead which cache partitioning introduces as memory gaps between arrays is comparable to the unused memory introduced within arrays with padding. However, these gaps between arrays enable

Table 1: Kernels and applications for experimental results

Name	Description	Lines of code	Num. loop sequences	Longest sequence	Maximum shift/peel
LL18	kernel from Livermore Loops	24	1	3	2/1
calc	kernel from qgbox [22] ocean model	186	1	5	3/3
filter	subroutine in hydro2d	247	1	10	5/4
tomcatv	SPEC95 benchmark (mesh generation)	190	1	3	1/1
hydro2d	SPEC95 benchmark (Navier-Stokes)	4292	3	10	5/4
spem	ocean circulation model [13]	26937	11	8	1/2

a predictable reduction in the number of misses, unlike the unpredictable outcome of padding.

5 Experimental Results

This section presents results of experiments conducted on a Convex Exemplar SPP-1000 multiprocessor and on a Kendall Square Research KSR2 multiprocessor to illustrate the performance advantages that can be obtained from our techniques. Both multiprocessors have performance monitoring hardware to measure execution time and the number of cache misses. The Convex is a more recent machine whose processors operate at 100 MHz and each have a 1-MByte direct-mapped cache. The KSR2 caches are two-way set-associative with a capacity of 256 KBytes, but the processors operate at only 40 MHz.

Our results are obtained using the three loop nest kernels and the three complete applications shown in Table 1. For each kernel or application, Table 1 also provides the number of loop nest sequences to which the shift-and-peel transformation was applied, as well as the length of the longest sequence and the maximum shift/peel amounts for any sequence. The loop nests of interest are transformed automatically with a prototype compiler implementation of the shift-and-peel transformation which is based on Polaris [7], a robust compiler framework from the University of Illinois at Urbana-Champaign. Accurate dependence distances are obtained with the Omega test [27], which are then used in the algorithm of Figure 8 to derive the required shifting and peeling. Fusion is then performed using the code manipulation facilities of Polaris. The calc sequence was analyzed and transformed manually because the inner loops in two of the loop nests prevents the current implementation from obtaining dependence distances.

The amounts of shifting and peeling for the three kernels are shown in detail in Table 2. These

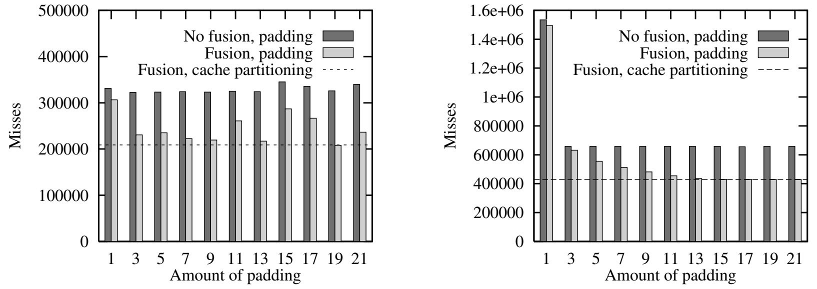
Table 2: Derived amounts of shifting and peeling

Loop	LL18 shifts	LL18 peels	calc shifts	calc peels	filter shifts	filter peels
1	0	0	0	0	0	0
2	1	0	0	0	0	0
3	2	1	2	2	0	0
4			3	3	1	1
5			3	3	2	2
6					2	2
7					3	3
8					4	4
9					4	4
10					5	4

results demonstrate that shift-and-peel is indeed required in order to fully exploit reuse between loop nests. It is important to note the dependences in these programs necessitate replication with the previous techniques of Callahan [8] and Appelbe and Smith [2] because *both* shifting and peeling are required, i.e., alignment conflicts exist. In contrast, our technique does not require any replication. Furthermore, the complexity of the dependence relationships in realistic loop nest sequences requires the systematic approach of the shift-and-peel transformation to automate its derivation and application. For instance, the dependence chain multigraph for `filter` contains 149 edges from which the shift and peel amounts in Table 2 are derived.

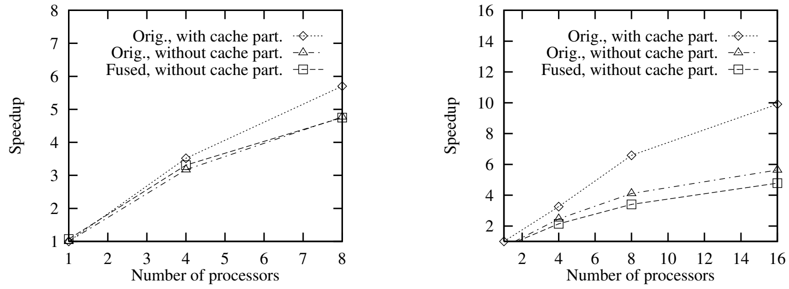
The remainder of the results to be presented in this section are organized as follows. First, we will demonstrate the importance of avoiding cache conflicts and the benefits of cache partitioning. Second, we will demonstrate the performance improvements obtained with our shift-and-peel transformation. Finally, we will compare our shift-and-peel transformation with the alignment/replication techniques of Callahan [8] and Appelbe and Smith [2]. All of the results we report below are actual measured performance on dedicated systems; they do not represent maximal or average performance.

The measured number of cache misses on a single processor during parallel execution for fused and unfused versions of the loops in `LL18` are shown in Figure 20. The number of misses obtained for various amounts of padding (shaded bars) is compared with the number of misses obtained from applying cache partitioning to the original arrays of size 512×512 (dashed line). The cache behavior with padding can be erratic, as discussed previously in Section 4. On the other hand,



(a) KSR2 (b) Convex

Figure 20: Cache partitioning for LL18

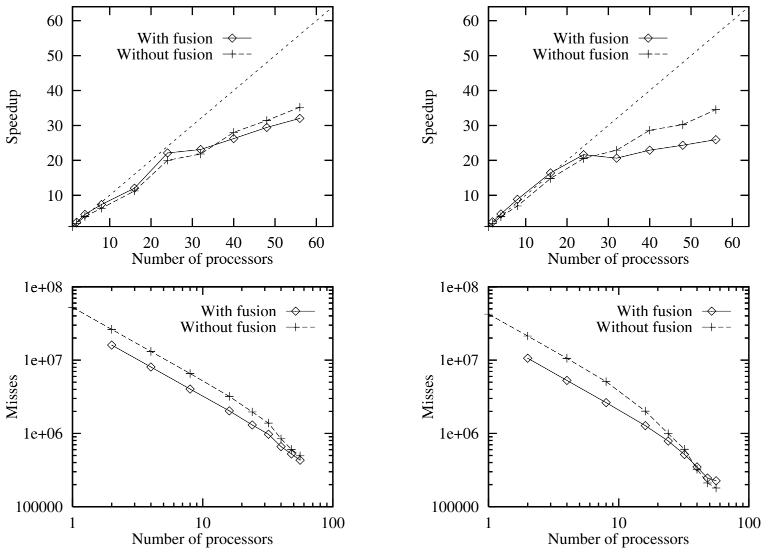


(a) hydro2d (b) tomcatv

Figure 21: Cache partitioning for applications on Convex

cache partitioning directly results in the smallest number of misses. It is interesting to observe that the potential benefit of fusion can be easily lost when padding fails to eliminate all conflict misses; the number of misses for the fused and unfused versions of the code are comparable for some amounts of padding. The results indicate that padding cannot guarantee the avoidance of conflicts after fusion.

The measured speedups for two of the applications (`hydro2d` and `tomcatv`) are shown with and without cache partitioning in Figure 21. The speedup of applying fusion with the shift-and-peel transformation in the absence of cache partitioning is also shown. The results indicate that cache conflict avoidance is necessary to ensure higher performance, not only for the original code, but also when applying a locality-enhancing transformation. The impact of conflicts on performance is greater for `tomcatv` because it is a smaller application where the loops in which the conflicts occur constitute a much larger fraction of the total execution time than `hydro2d`. Nonetheless, performance is degraded in both cases by the occurrence of conflicts. In light of these results, *our subsequent comparisons are based on cache-partitioned memory layout*, and hence represent a *lower bound* on the performance improvement attainable with the shift-and-peel transformation.

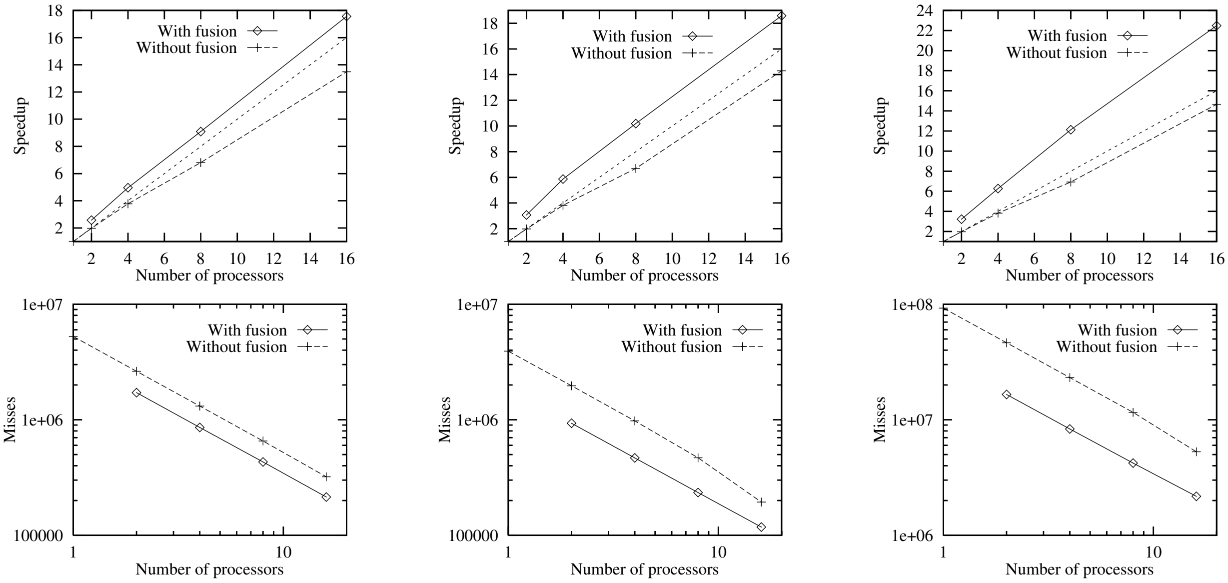


(a) LL18 (b) calc

Figure 22: Speedup and misses of kernels on KSR2

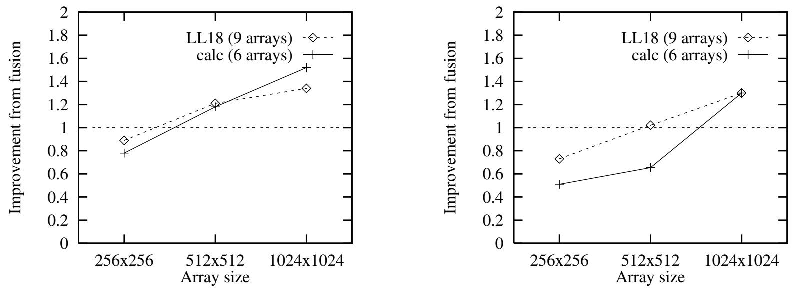
The next set of results show the parallel performance of fused and unfused versions of `LL18` and `calc` using cache-partitioned memory layout for up to 56 processors on the KSR2. All array sizes were 512×512. For each parallel run (with or without fusion), the speedup is relative to the unfused version executing on a single processor (again, using cache-partitioned layout as mentioned above). Figure 22 shows the speedups and misses on the KSR2. For `LL18`, fusion improves performance by 7% to 15% for up to 32 processors, beyond which the unfused version performs better. Similarly, fusion improves the performance for `calc` by 6% to 20% up to 24 processors, beyond which the unfused version performs better. It is important to note that the unfused versions are already benefitting from cache partitioning, hence these results reflect the benefit from fusion alone. As such, they provide a lower bound on the performance improvement.

Figure 23 shows the parallel performance of fused and unfused versions of the kernels on the Convex. Array sizes were 1024×1024 for `LL18` and `calc`, and 1602×640 for `filter`. As before, speedup is computed with respect to the unfused version of each kernel on a single processor, and cache partitioning is used in all the experiments. Fusion improves performance by at least 30% for `LL18` and `calc`, and by 60% for `filter`. Locality enhancement results in greater improvements on the Convex because of its higher cache miss penalty.



(a) `LL18` (b) `calc` (c) `filter`

Figure 23: Speedup and misses of kernels on Convex

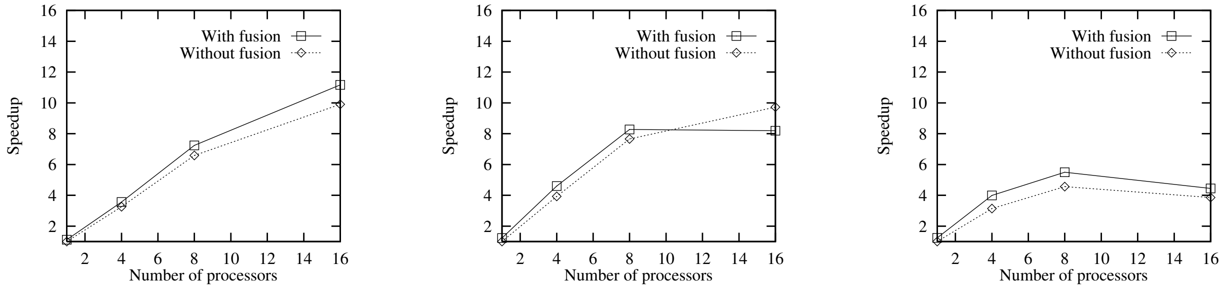


(a) 8 processors (b) 16 processors

Figure 24: Improvement from fusion for `LL18` and `calc` on Convex

The results on the KSR2 (Figure 22) indicate that for a fixed data size, the benefit of fusion diminishes as the number of processors increases, until eventually the fused version performs worse than the unfused version. As the number of processors is increased for a *fixed* problem size, the portion of the data used by *each* processor begins to fit in its cache. This reduces the need for, and benefit of, locality-enhancing transformations. As a result, the overhead of the shift-and-peel transformation outweighs its benefit, degrading performance. Since `LL18` uses nine arrays, while `calc` uses only six, locality enhancement for `LL18` is beneficial for a larger number of processors than for `calc`. These observations suggest using knowledge of data sizes and cache sizes to determine the profitability of applying the shift-and-peel transformation for locality enhancement.

To further study the implications of data size with respect to cache size on performance, the

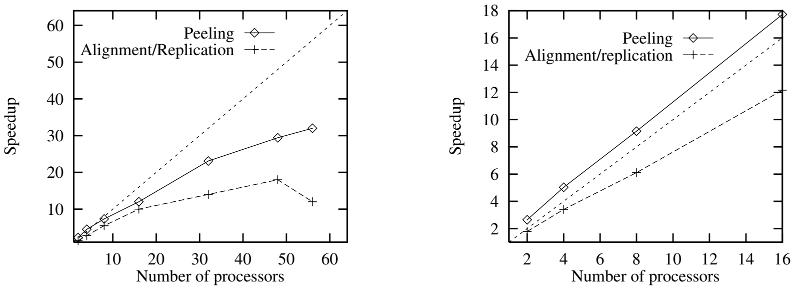


(a) `tomcatv` (b) `hydro2d` (c) `spem`

Figure 25: Speedup for applications on Convex

array sizes in `LL18` and `calc` were varied. The results are shown in Figure 24 for the Convex; the horizontal axes represent different array sizes, and the vertical axes represent the relative performance of fusion, which is computed as the ratio of execution times of the unfused and fused loops, respectively. Any point above the reference line at 1 indicates that fusion improves performance. As before, cache partitioning is used throughout, hence any improvements represent lower bounds. Figure 24(a) indicates that with 8 processors, the two larger array sizes are such that the data does not entirely fit in the cache, hence fusion improves performance. In particular, the improvement is up to 50% in the case of `calc`. With 16 processors, the total cache capacity is doubled, and Figure 24(b) indicates that even the 512×512 array size permits data to fit in the cache for `calc`, hence fusion does not improve performance. Note that because `LL18` has more arrays of the same size than `calc`, fusion still improves performance when the array size is 512×512 because all the data cannot simultaneously fit in the caches.

The benefit of the shift-and-peel transformation with cache partitioning for complete applications on the Convex is shown in Figure 25. For `tomcatv`, the array size is 513×513, and the total data size is 16 Mbytes. For `hydro2d`, the array size is 802×320, and the total data size is 50 Mbytes. For `spem`, the array size is $60 \times 65 \times 65$, and the total data size is 70 Mbytes. The speedups in Figure 25 are relative to the original code with cache partitioning on one processor. For `tomcatv`, fusion consistently improves parallel performance by 10% to 12%. For `hydro2d`, the benefit of the shift-and-peel transformation is 23% on one processor, and diminishes to 8% on eight processors. At 16 processors, the data begins to fit in the caches, so the overhead of the shift-and-peel transformation limits the fused speedup. The improvement for `spem` is consistently



(a) KSR2 (b) Convex

Figure 26: Performance of peeling and alignment/replication for `LL18`

20% up to eight processors because this application had the largest number of transformed loop sequences, and these sequences constitute close to half of total execution time. However, at 16 processors, remote memory accesses cause the speedup for both the fused and unfused versions to fall below the speedup at 8 processors. This behavior results from the Convex compiler transforming and parallelizing loops without regard for potential increases in remote memory traffic.

The final set of results compares our shift-and-peel transformation with the alignment and replication techniques proposed by Callahan [8] and Appelbe and Smith [2]. Figure 26 compares the performance of the fused `LL18` loop nest parallelized using peeling with the performance of the same loop nest parallelized using direct application of alignment and replication. In this case, it was necessary to replicate two arrays and two statements to enable synchronization-free parallel execution. The figure clearly indicates that superior performance is achieved using peeling. This is attributed to the overhead associated with the replication of code and data.

6 Conclusions

In this paper, we presented systematic techniques to enhance cache locality in data-parallel programs. We first presented the shift-and-peel transformation for fusing a sequence of parallel loop nests. It consists of: (1) a shifting transformation which enables fusion, even in the presence of fusion-preventing dependences; and (2) a peeling transformation which overcomes loop-carried dependences in the resulting fused loop nest. We also presented cache partitioning as a method to eliminate cache conflicts among array elements in a predictable manner.

We have described a framework for the shift-and-peel transformation which permits its systematic derivation and application. The shift-and-peel transformation enhances locality while preserving existing parallelism without the overhead of data and/or code replication necessary in previous techniques. Likewise, cache partitioning is also systematic in that it determines a memory layout which avoids conflicts, obviating the need for padding heuristics.

We have evaluated our techniques using both kernels and complete applications on two different multiprocessors. Performance measurements indicate that our techniques improve parallel performance by up to 60% for kernels and up to 20% for applications. The benefit of our techniques increases with faster processors due to higher cache miss latencies. With the growing gap between processor and memory speeds, we expect our techniques to result in greater performance improvements on future multiprocessor systems.

Experimental results indicate that performance tradeoffs do exist. Performance gains resulting from shift-and-peel diminish as the number of processors increases because the portion of data used by each processor is more likely to fit in its cache. When the number of processors is sufficiently large, the overhead of shift-and-peel outweighs its benefit, and performance losses result. Hence, the profitability of the transformation should be evaluated in the compiler with knowledge of the data size with respect to the cache size.

Acknowledgements

This research is supported by grants from the Natural Sciences and Engineering Research Council of Canada and the Information Technology Research Centre of Ontario. The use of the KSR2 and Convex SPP-1000 multiprocessors was provided by the University of Michigan Center for Parallel Computing. The authors are grateful for the assistance provided by Andrew Caird of the Center for Parallel Computing. The authors would also like to thank the anonymous referees for their useful comments and suggestions.

References

[1] A. Agarwal et al. The MIT Alewife Machine: A large-scale distributed-memory multiprocessor. In Michel Dubois and Shreekant Thakkar, editors, *Scalable shared memory multiprocessors*, pages 239–261. Kluwer Academic Publishers, Boston, 1992.

[2] W. Appelbe and K. Smith. Determining transformation sequences for loop parallelization. In U. Banerjee et al., editors, *Languages and Compilers for Parallel Computing*, pages 208–222. Springer-Verlag, 1993.

[3] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26:345–420, December 1994.

[4] D. F. Bacon et al. A compiler framework for restructuring data declarations to enhance cache and TLB effectiveness. In *CASCON'94*, pages 270–282, Toronto, Canada, 1994.

[5] U. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Boston, 1988.

[6] U. Banerjee, R. Eigenmann, A. Nicolau, and D. A. Padua. Automatic program parallelization. *Proceedings of the IEEE*, 81(2):211–243, February 1993.

[7] B. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, B. Pottenger, L. Rauchwerger, P. Tu, and S. Weatherford. Polaris: Improving the effectiveness of parallelizing compilers. In K. Pingali et al., editors, *Languages and Compilers for Parallel Computing*, pages 141–154. Springer-Verlag, Berlin, 1995.

[8] C. D. Callahan. *A Global Approach to the Detection of Parallelism*. PhD thesis, Dept. of Computer Science, Rice University, March 1987.

[9] Convex Computer Corporation. *Convex Exemplar system overview*. Richardson, TX, 1994.

[10] Cray Research GmbH. *The Cray Research massively parallel processor system — Cray T3D*. 80922 Munchen, Germany, 1993.

[11] A. Ganesh. Fusing loops with backwards inter loop data dependence. *ACM SIGPLAN Notices*, 29(12):25–30, December 1994.

[12] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformation. *J. of Parallel and Distributed Computing*, 5:587–616, 1988.

[13] K. S. Hedstrom. User's Manual for a Semi-spectral Primitive Equation Ocean Circulation Model–Version 3.9. Technical report, Institute of Marine and Coastal Sciences, Rutgers University, March 1994.

[14] M. Heinrich et al. The Stanford FLASH multiprocessor. In *Proc. 21th Intl. Symp. on Computer Architecture*, pages 302–313, Chicago, IL., April 1994.

[15] Kendall Square Research. *KSR1 Principles of operation*. Waltham, MA, 1991.

[16] K. Kennedy and K. S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In U. Banerjee et al., editors, *Languages and Compilers for Parallel Computing*, pages 301–320. Springer-Verlag, Berlin, 1994.

[17] M. Lam, E. Rothberg, and M. Wolf. The cache performance and optimizations of blocked algorithms. In *Proc. ASPLOS-IV*, pages 63–74, Santa Clara, CA., 1991.

[18] A. Lebeck and D. Wood. Cache profiling and the SPEC benchmarks: A case study. *IEEE Computer*, 27(10):15–26, 1994.

[19] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. The Stanford Dash Multiprocessor. *Computer*, 25(3):63–79, March 1992.

[20] N. Manjikian and T. Abdelrahman. Reduction of cache conflicts in loop nests. Tech. Rep. CSRI-318, Computer Systems Research Institute, University of Toronto, Canada, March 1995.

[21] N. Manjikian and T. Abdelrahman. Scheduling of Wavefront Parallelism on Scalable Shared-memory Multiprocessors. In *Proceedings of the 1996 International Conference on Parallel Processing*, pages III121–III123, Bloomingdale, IL., August 1996.

[22] J. McCalpin. Quasigeostrophic box model–revision 2.3. Technical report, College of Marine Studies, University of Delaware, 1992.

[23] D. A. Padua and M. J. Wolfe. Advanced compiler optimizations for supercomputers. *Comm. ACM*, 29(12):1184–1201, December 1986.

[24] N. Passos and E. Sha. Full Parallelism in Uniform Nested Loops Using Multi-dimensional Retiming. In *Proceedings of the 1994 International Conference on Parallel Processing*, pages II130–II133, August 1994.

[25] A. K. Porterfield. *Software Methods for Improvement of Cache Performance on Supercomputer Applications*. PhD thesis, Dept. of Computer Science, Rice University, April 1989.

[26] W. Pugh. Uniform techniques for loop optimization. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, pages 341–352, Cologne, Germany, June 1991.

[27] W. Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):102–114, August 1992.

[28] Z. Vranesic et al. The NUMAchine multiprocessor. Tech. Rep. CSRI-324, Computer Systems Research Institute, University of Toronto, Canada, April 1995.

[29] Z. Vranesic, M. Stumm, D. Lewis, and R. White. Hector: A hierarchically structured shared-memory multiprocessor. *IEEE Computer*, 24(1):72–79, January 1991.

[30] J. Warren. A hierarchical basis for reordering transformations. In *Proc. 11th ACM Symposium on the Principles of Programming Languages*, pages 272–282, June 1984.

[31] M. E. Wolf. *Improving Locality and Parallelism in Nested Loops*. PhD thesis, Department of Computer Science, Stanford University, 1992.

[32] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proc. ACM Conference on Programming Language Design and Implementation*, pages 30–44, Toronto, Canada, 1991.

[33] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers*. The MIT Press, Cambridge, MA., 1989.

Appendix I: Legality Proof for the Shift-and-peel Transformation

For simplicity, this proof is presented for sequences of parallel one-dimensional loops with identical loop bounds. Generalization to perfectly-nested multidimensional loop nests with differing loop bounds is straightforward, but tedious. First, a number of definitions are provided.

Definition 1 *A sequence of loops $L_1, \ldots, L_n$ is an admissible* parallel loop sequence *if there is no intervening code between the loops, if each loop L_k $(1 \leq k \leq n)$ is parallel, and if all loops use the same integer index variable* `I` *with the same integer lower/upper bounds ℓ and u ($\ell \leq u$) and a step of 1. The loop sequence is totally ordered, i.e., $L_1 \prec L_2 \prec \cdots \prec L_n$. The computation performed for an iteration* `I`$=i$ *($\ell \leq i \leq u$) within the body of a loop L_k $(1 \leq k \leq n)$ is denoted by $S_k(i)$.*

Definition 2 *For a loop L_k in a parallel loop sequence $L_1, \ldots, L_n$, the set of all memory locations read in a given iteration i of the loop body $S_k(i)$ is denoted by $\mathcal{R}_k(i)$. Similarly, the set of all memory locations written in a given iteration i of the loop body $S_k(i)$ is denoted by $\mathcal{W}_k(i)$.*

Definition 3 *For a pair of loops L_a, L_b in a parallel loop sequence $L_1, \ldots, L_n$, where $L_a \prec L_b$, an* interloop dependence *$S_a(i_1)\delta S_b(i_2)$ exists between iteration i_1 in L_a and iteration i_2 in L_b if*

$$[\mathcal{R}_a(i_1) \cap \mathcal{W}_b(i_2) \neq \emptyset] \vee [\mathcal{W}_a(i_1) \cap \mathcal{R}_b(i_2) \neq \emptyset] \vee [\mathcal{W}_a(i_1) \cap \mathcal{W}_b(i_2) \neq \emptyset] \,,$$

where $\ell \leq i_1 \leq u$ and $\ell \leq i_2 \leq u$. The dependence distance is given by $i_2 - i_1$, and may be positive, negative, or zero.

Definition 4 *Let L_a and L_b denote a pair of loops in a parallel loop sequence $L_1, \ldots, L_n$ such that $L_a \prec L_b$. Let $\mathit{DEP}_{a,b}$ denote the set of all interloop dependences $S_a(i_1)\delta S_b(i_2)$ between L_a and L_b. Let $\mathit{DEP}_{a,b}(d)$ denote the subset of all interloop dependences between the loops L_a and L_b with distance d. Let $\mathit{DIST}_{a,b}$ denote the set of all distances d such that $\mathit{DEP}_{a,b}(d) \neq \emptyset$. $\mathit{DEP}_{a,b}$ is a set of* uniform interloop dependences *if:*

$$\forall d \in \mathit{DIST}_{a,b},\ \ \exists S_a(i)\delta S_b(i+d) \in DEP_{a,b}(d),\ \ \begin{cases} \forall \ell \leq i \leq u-d, & \text{if } d \geq 0, \\ \forall \ell - d \leq i \leq u, & \text{if } d < 0. \end{cases}$$

Uniformity requires interloop dependences with distance d to flow from all *iterations i in L_a to $i+d$ in L_b, subject to the loop bound constraints.*

Definition 5 *For a parallel loop sequence $L_1, \ldots, L_n$ in which all interloop dependences are uniform, let $\mathit{shift}(k) \leq 0$ and $\mathit{peel}(k) \geq 0$ denote the amounts of shifting and peeling derived for each loop L_k $(1 \leq k \leq n)$ by the shift-and-peel derivation algorithm. Let P denote the number of processors to be used for parallel execution. Let $\mathit{istart}(p)$ and $\mathit{iend}(p)$ denote the start and end for a subset of consecutive iterations from the original iteration space bounded by ℓ and u to be executed by a processor p $(1 \leq p \leq P)$, i.e.,*

$$\mathit{istart}(p) = \ell + \left\lfloor \frac{u - \ell + 1}{P} \right\rfloor \cdot (p-1), \quad \mathit{iend}(p) = \begin{cases} \mathit{istart}(p) + \left\lfloor \dfrac{u-\ell+1}{P} \right\rfloor - 1, & 1 \leq p < P, \\ u, & p = P. \end{cases}$$

The shift-and-peel transformation yields a fused loop whose iterations are executed in parallel on P processors, a synchronization point, and a number of peeled loop iterations which are also executed in parallel on P processors. For each processor p $(1 \leq p \leq P)$, FUSED(p) is the the subset of iterations from the fused loop, i.e.,

$$\textit{FUSED}(p) = \begin{cases} \{S_k(i) \mid \textit{istart}(p) \leq i \leq \textit{iend}(p) + \textit{shift}(k),\ \ 1 \leq k \leq n\}, & p = 1, \\ \{S_k(i) \mid \textit{istart}(p) + \textit{peel}(k) \leq i \leq \textit{iend}(p) + \textit{shift}(k),\ \ 1 \leq k \leq n\}, & 1 < p \leq P. \end{cases}$$

Similarly, PEELED(p) is the subset of peeled iterations for a processor p, i.e.,

$$\textit{PEELED}(p) = \begin{cases} \{S_k(i) \mid \textit{iend}(p) + \textit{shift}(k) + 1 \leq i \leq \textit{iend}(p) + \textit{peel}(k), 1 \leq k \leq n\}, & 1 \leq p < P, \\ \{S_k(i) \mid \textit{iend}(p) + \textit{shift}(k) + 1 \leq i \leq \textit{iend}(p), 1 \leq k \leq n\}, & p = P. \end{cases}$$

Definition 6 *For a parallel loop sequence $L_1, \ldots, L_n$ in which all interloop dependences are uniform, let shift$(k) \leq 0$ and peel$(k) \geq 0$ denote the amounts of shifting and peeling derived for each loop L_k $(1 \leq k \leq n)$ by the shift-and-peel derivation algorithm. The* iteration count threshold *N_t for the parallel loop sequence is defined as*

$$N_t = \max_{1 \leq k \leq n} \textit{peel}(k) - \textit{shift}(k).$$

With the preceding set of definitions, the following theorem on the legality of the shift-and-peel transformation may now be proved.

Theorem 1 *The shift-and-peel transformation is always legal for a parallel loop sequence $L_1, \ldots, L_n$ in which all interloop dependences are uniform, provided that*

$$\left\lfloor \frac{u - \ell + 1}{P} \right\rfloor \geq N_t,$$

where P is the number of processors used in parallel execution of the resulting loop, $u - \ell + 1$ is the number of iterations in each of the loops of the original parallel loop sequence, and N_t is the iteration count threshold in Definition 6.

Proof First, all of the original computation is performed in the transformed code. Using Definition 5, this condition is satisfied by noting that the lower and upper bounds for the fused loop in each processor together with the peeled iterations cover the original computation, i.e.,

$$\bigcup_{1 \leq p \leq P} (\textit{FUSED}(p) \cup \textit{PEELED}(p)) = \{S_k(i) \mid \ell \leq i \leq u,\ \ 1 \leq k \leq n\}.$$

Second, there is no redundancy in the computation. Each element of the original computation is performed by exactly one processor, provided that

$$\left\lfloor \frac{u - \ell + 1}{P} \right\rfloor \geq N_t.$$

Each processor performs consecutive subsets of iterations from each of the original loops. To ensure no redundant computation is performed by two adjacent processors p and $p + 1$, it must be true that

$$\textit{PEELED}(p) \cap \textit{PEELED}(p+1) = \emptyset,$$

for which Definition 5 implies that the following condition must be satisfied:

$$\forall 1 \le p < P, \forall 1 \le k \le n, \textit{iend}(p) + \textit{peel}(k) < \textit{iend}(p+1) + \textit{shift}(k) + 1.$$

Substitution for $\textit{iend}(p)$ and $\textit{iend}(p+1)$ using Definition 5 and simplification yields

$$\forall 1 \le k \le n, \left\lfloor \frac{u - \ell + 1}{P} \right\rfloor + 1 > \textit{peel}(k) - \textit{shift}(k).$$

Since it must be true for all loops, it must be true for the loop for which $\textit{peel}(k) - \textit{shift}(k)$ is the largest, and this is given by the iteration count threshold N_t. Since both $\left\lfloor \frac{u-\ell+1}{P} \right\rfloor$ and N_t are integers, the condition may be simplified to

$$\left\lfloor \frac{u - \ell + 1}{P} \right\rfloor \ge N_t.$$

With this condition satisfied, it can also be shown using Definition 5 that $\textit{FUSED}(p) \cap \textit{FUSED}(p+1) = \emptyset$, $\textit{PEELED}(p) \cap \textit{FUSED}(p+1) = \emptyset$, and $\textit{PEELED}(p+1) \cap \textit{FUSED}(p) = \emptyset$, $\forall 1 \le p < P$.

Third, none of the original uniform interloop dependences are violated when the $\textit{FUSED}(p)$ subsets are executed in parallel on P processors. For interloop dependences $S_a(i)\delta S_b(i+d)$ such that $S_a(i), S_b(i+d) \in \textit{FUSED}(p)$ and $d < 0$, shifting trivially ensures that the dependences are satisfied internally within each subset. Dependences with $d > 0$ are always satisfied internally even with shifting. Furthermore, no dependences flow between the $\textit{FUSED}(p)$ subsets executed in parallel on different processors. This is easily shown with a proof by contradiction.

For $S_a(i)\delta S_b(i+d)$, assume that $S_a(i) \in \textit{FUSED}(p_1)$ and $S_b(i+d) \in \textit{FUSED}(p_2)$, where $p_1 \ne p_2$. For $d \ge 0$, assume that $p_2 > p_1$. The shift-and-peel derivation algorithm will result in $\textit{peel}(b) \ge d$. If $S_a(i) \in \textit{FUSED}(p_1)$, the maximum value of i is $i = \textit{iend}(p_1) + \textit{shift}(a)$ by Definition 5. If $S_b(i+d) \in \textit{FUSED}(p_2)$, where $p_2 > p_1$, then by Definition 5, it must be true that

$$i + d = \textit{iend}(p_1) + \textit{shift}(a) + d \ge \textit{istart}(p_1 + 1) + \textit{peel}(b).$$

Substitution for $\textit{iend}(p_1)$ and $\textit{istart}(p_1 + 1)$ from Definition 5, with some rearrangement, yields

$$d - 1 + \textit{shift}(a) \ge \textit{peel}(b).$$

Since $\textit{shift}(a) \le 0$, and $\textit{peel}(b) \ge d$,

$$d - 1 \ge d - 1 + \textit{shift}(a) \ge \textit{peel}(b) \ge d.$$

But $d - 1 < d$, hence this is a contradiction. Since the *maximum* iteration i such that $S_a(i) \in \textit{FUSED}(p_1)$ was used, this contradiction is true for all iterations i such that $S_a(i) \in \textit{FUSED}(p_1)$. A similar contradiction results from assuming that $p_2 < p_1$ for $d < 0$. Thus, no dependences flow between the $\textit{FUSED}(p)$ subsets for any pair of processors.

Fourth, none of the original uniform interloop dependences are violated when the $\textit{PEELED}(p)$

subsets are executed in parallel on P processors. Internally, any interloop dependences $S_a(i)\delta S_b(i+d)$ such that $S_a(i), S_b(i+d) \in$ *PEELED*(p) are always satisfied because all iterations within the subset which are peeled from loop L_a are executed before iterations peeled from L_b. Furthermore, no dependences flow between *PEELED*(p) subsets executed on different processors. This is easily shown with a proof by contradiction similar to that used above for the *FUSED*(p) subsets.

Finally, none of the original uniform interloop dependences are violated across the synchronization point between the execution of *FUSED*(p) and *PEELED*(p) on each processor because all interloop dependences either flow interally within each fused or peeled subset of iterations, or from a fused subset to a peeled subset. The total ordering implies that all dependences flow forward in the original sequence. For those dependences which require peeling, it is always the sink iteration which is peeled. Any other iterations which depend on a peeled iteration are also peeled by virtue of the shift-and-peel derivation algorithm. Thus, dependences never flow from a peeled subset to a fused subset. The synchronization point ensures that those dependences flowing from a fused subset to a peeled subset are always satisfied.

Since the transformed code executes all of the original computation without redundancy on P processors (provided that the iteration count threshold condition is satisfied), and none of the original interloop dependences are violated internally within the subsets of iterations executed by different processors or externally between the subsets, the shift-and-peel transformation is legal. $\square$