Reduction of Cache Conflicts in Loop Nests

Naraig Manjikian and Tarek Abdelrahman

Technical Report CSRI-318
March 1995

Computer Systems Research Institute
University of Toronto
Toronto, Canada
M5S 1A1

The Computer Systems Research Institute (CSRI) is an interdisciplinary group formed to conduct research and development relevant to computer systems and their application. It is an Institute within the Faculty of Applied Science and Engineering, and the Faculty of Arts and Science, at the University of Toronto, and is supported in part by the Natural Sciences and Engineering Research Council of Canada.

Reduction of Cache Conflicts in Loop Nests

Naraig Manjikian and Tarek Abdelrahman
Department of Electrical and Computer Engineering
The University of Toronto
Toronto, Ontario, Canada M5S 1A4

Abstract

We address the problem of cache conflicts in loop nests. Cache conflicts degrade performance, particularly for locality-enhancing transformations, which rely on retaining reusable data in the cache to improve performance. We present a new technique called *cache partitioning* which eliminates conflicts by logically dividing the cache into a number of partitions and adjusting the array layout in memory to map different arrays to different partitions. One-dimensional cache partitioning, in which partitions contain contiguous data from each array, is described first, followed by multidimensional partitioning, in which the data in each partition need not be contiguous. The elimination of conflicts with cache partitioning enables accurate modelling of the benefit of locality-enhancing transformations. A new model is presented which is based on the *sweep ratio*, defined as the ratio of the amounts of data traffic between the cache and memory before and after applying a locality-enhancing transformation. The performance benefits of two important locality-enhancing transformations, loop fusion and tiling, are modelled. Results of experiments conducted on a KSR machine demonstrate that cache partitioning consistently eliminates conflicts, and does so directly without the trial-and-error needed for techniques such as array padding. The results also validate the performance model and demonstrate that cache partitioning allows the full benefit of both transformations to be realized.

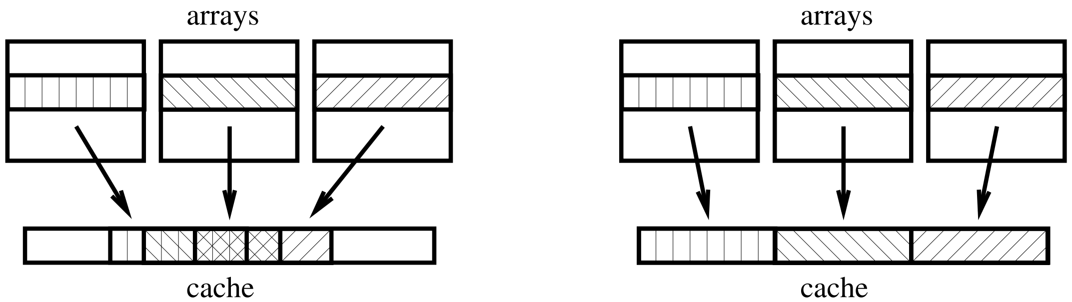
Index Terms — Cache conflicts, Data layout, Data reuse, Locality enhancement, Loop transformations, Optimizing compilers

1 Introduction

Cache conflicts among arrays during the execution of loop nests in scientific programs cause data to be ejected from the cache, leading to cache misses and degrading performance. The occurrence of these conflicts is sensitive to cache hardware parameters, array sizes, and data access patterns in a loop nest. It is difficult to guarantee consistently high levels of performance since small changes in these factors may cause the number of conflicts to increase sharply. Conflicts are particularly undesirable when locality-enhancing transformations such as loop fusion [1] and tiling [10] are used, since these transformations rely on keeping reusable data in the cache. Such transformations change data access patterns, increasing the likelihood of conflicts and making it more difficult to resolve them effectively [5]. Consequently, the benefit of enhancing locality can be diminished. Furthermore, conflicts prevent compilers from accurately assessing the profitability of applying such transformations.

We present a new technique called *cache partitioning* which eliminates array conflicts within a loop nest. The key idea behind the technique is to logically divide the cache into a number of partitions and adjust the array layout in memory to map different arrays to different partitions. Cache partitioning offers a number of advantages compared to previous techniques that address the problem of conflicts, such as array padding and memory alignment [1]. Cache partitioning consistently eliminates conflicts, and does so directly without the trial-and-error needed to determine a padding factor or the necessary alignment. The elimination of conflicts enables accurate modelling of the benefit of locality-enhancing transformations. This is particularly desirable as it permits compilers to more accurately assess the profitability of such transformations. Cache partitioning fully utilizes the available cache capacity, which in some cases can simplify the choice of parameters for locality-enhancing transformations.

In this paper, cache partitioning is described and its benefit for a number of applications is assessed. Experimental evidence demonstrating how cache partitioning consistently eliminates conflicts is provided and this consistency is contrasted with the erratic results obtained



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

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

Figure 1: Avoiding conflicts in the cache

with padding. The performance benefits of two important locality-enhancing transformations, loop fusion and tiling, are modelled assuming the elimination of conflicts with cache partitioning. Predictions from the model are compared with measured performance improvements to validate the model and to demonstrate that cache partitioning allows the full benefit of each transformation to be realized.

This paper is organized as follows. In Section 2, we present the details for applying cache partitioning for arrays referenced in one or more loop nests. In Section 3, we describe our analytical performance model. In Section 4, we discuss the application of cache partitioning to fusion and tiling. In Section 5, we present experimental results. In Section 6, we discuss related work. Finally, in Section 7, we conclude and give directions for future research.

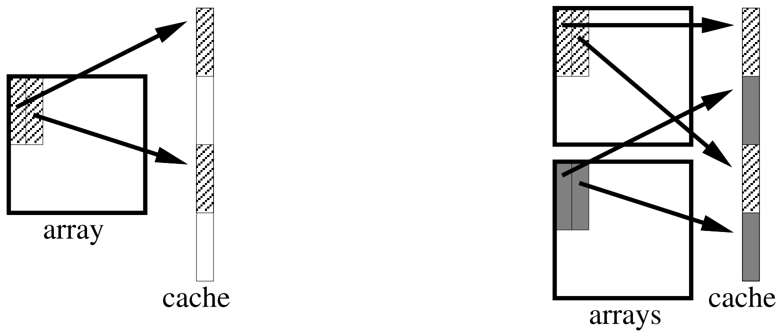
2 Cache Partitioning

Conflicts occur when data from different arrays maps into overlapping regions of the cache during the execution of a loop nest, as in Figure 1(a). Cache partitioning removes these conflicts by adjusting the memory layout of the arrays. The basic idea is to logically partition the cache into nonoverlapping regions, one for each array, and then adjust the starting addresses of the arrays in memory such that data from each array maps into a different partition, as in Figure 1(b). The starting addresses of the arrays are adjusted by inserting appropriately-sized gaps between the arrays in memory. Note that cache partitioning only affects the array starting addresses; no changes are required to the array index expressions.

Cache partitioning ensures that reusable data is not ejected from the cache due to conflicts. It is common to assume that exploitable reuse for references to a given array in a loop nest exists only if those references generate uniform dependences [3, 10]; we also make this assumption. Given a loop nest with dimensionality ℓ, pairs of references to an array A with dimensionality k can be represented as $A(f(\vec{\imath})), A(g(\vec{\imath}))$, where $\vec{\imath} = (i_1, i_2, \ldots, i_\ell)^T \in \mathbf{Z}^\ell$. The dependences for A are uniformly generated if $f(\vec{\imath}) = h_A \cdot \vec{\imath} + c_f$ and $g(\vec{\imath}) = h_A \cdot \vec{\imath} + c_g$, where $h_A : \mathbf{Z}^\ell \mapsto \mathbf{Z}^k$ is a linear mapping from the iteration space to the data space represented by a $k \times \ell$ matrix, and c_f, c_g are constant vectors in $\mathbf{Z}^k$. Uniformly generated dependences translate directly to spatial and temporal reuse of array data which can be effectively exploited in cache. Little exploitable reuse exists in the absence of this uniformity [10].

Cache partitioning is effective when data access patterns across the arrays in a loop nest are *compatible*, i.e., have the same stride and direction with respect to each loop level. Hence, compatibility requires that $h_A = h_B$ for pairs of arrays A and B generating uniform dependences in a loop nest. With compatibility, a conflict-free mapping of array starting addresses into the cache ensures that the mapping for the remaining array data will also be conflict-free. As the loop nest is executed, new data from different arrays is brought into the cache and the partition boundaries effectively move through cache, but the reusable portions of different arrays in the partitions themselves never overlap.

Although compatibility seems restrictive, it is often satisfied in the presence of exploitable reuse. Furthermore, it is possible to apply straightforward transformations to obtain collective compatibility from uniform dependences. 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 [6] 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. Often, such transformations are needed anyway to improve the utilization of cache lines, i.e., to effectively exploit spatial reuse.



(a) Reducing the number of indices from an inner dimension and the resulting noncontiguity

(b) Interleaving noncontiguous partitions to prevent conflicts between different arrays into the cache

Figure 2: Multidimensional cache partitioning

When partitioning the cache for a number of multidimensional arrays, the limited cache capacity does not allow the resulting partitions to hold the entire contents of each array. Consequently, it is necessary to reduce the number of indices from one or more array dimensions which are simultaneously resident in a partition. The simplest case is one-dimensional cache partitioning, where a restriction is placed on the number of indices from the outermost array dimension. All indices from the remaining inner dimensions for a given index of the outermost dimension are resident in a partition simultaneously. As a result, the data from each array is contiguous, and so are the cache partitions containing that data.

More generally, the limit on the number of indices from the outermost dimension may be too small. Hence, it is necessary to perform multidimensional cache partitioning, which reduces the number of indices from other array dimensions and makes additional cache space available to accommodate a larger number of indices from the outermost dimension. In this case, the data in each partition is no longer contiguous because reducing the number of indices from inner dimensions skips over portions of the array in memory. Since the data is not contiguous in memory, the partitions containing this data in the cache are not contiguous either, as shown in Figure 2(a). These noncontiguous partitions must be carefully interleaved in the cache to ensure that they do not overlap and cause conflicts, as in Figure 2(b).

```
GreedyMemoryLayout(A)::                                      // A = set of arrays
 n_a = |A|
 s_p = c/n_a                                                  // partition size
 C = {0, s_p, 2·s_p, ..., (n_a − 1)·s_p}                      // partition starting addresses
 P = {0, 1, ..., n_a − 1}                                     // available partition indices
 q = q_0                                    // 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 partiitons
     gap(p) = C(p)− mapped_cache_address
     if C(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 3: Greedy memory layout algorithm for cache partitioning

2.1 One-dimensional Cache Partitioning

In one-dimensional partitioning, the array dimension along which to partition is the outermost one, as indicated above. This approach results in contiguous partitions containing contiguous data from each array. Given n_a arrays with dimensions $N_1 \times N_2 \times \cdots \times N_k$[1], and a cache capacity of c elements, the size of each partition is $s_p = \lfloor c/n_a \rfloor$ elements. Without loss of generality, we assume column-major storage order, i.e., elements in a column are stored contiguously in memory. The N_1 elements in the first dimension comprise a column and are stored contiguously. The outermost array dimension is k, hence each partition contains a contiguous block of data with dimensions $N_1 \times N_2 \times \cdots \times N_{k-1} \times B_k$, where the B_k is the limit on the number of indices from the outermost dimension, and is given by $B_k = \lfloor s_p/(N_1 \cdot N_2 \cdots N_{k-1}) \rfloor$.

Within the cache, the n_a partitions must be separated by a distance s_p to ensure that

[1] We assume for simplicity that all arrays are of equal size. This is typically the case in most applications. The technique can be extended to accommodate arrays of different sizes.

they do not overlap. If the first partition begins at address 0 in the cache, the starting addresses in the cache for the partitions are $0, s_p, 2 \cdot s_p, \ldots, (n_a - 1) \cdot s_p$. The starting address of each array in memory must be adjusted such that it maps to one of the valid partition starting addresses in the cache. The adjustment of array starting addresses is accomplished by inserting gaps between arrays in memory. These gaps represent memory overhead which should be minimized. The greedy layout algorithm shown in Figure 3 is employed to reduce the size of these gaps when mapping arrays to partitions. The arrays are selected in an arbitrary order. A set of available partitions P is maintained, and each array to be placed in memory is assigned to a cache partition of size s_p which minimizes the distance between the starting address required for that partition and the end of the array most recently placed in memory. Although multiple memory addresses map into the selected partition, the address in free memory closest to the end of the most recently placed array is always used. Each partition selected in this manner is removed from the set of available partitions 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 $\textsc{CacheMap}()$. The complexity of the algorithm is $O(n_a^2)$. The overhead from the gaps introduced by this algorithm (measured by the increase in the amount of memory required) is bounded by

$$o_m < \frac{2 \cdot c}{n_a \cdot d},$$

where c is the cache size, and $d = N_1 \cdot N_2 \cdots N_k$, the size of each array. The overhead diminishes rapidly as the data size increases relative to the cache size, which is the case in applications to which locality optimizations are applied.

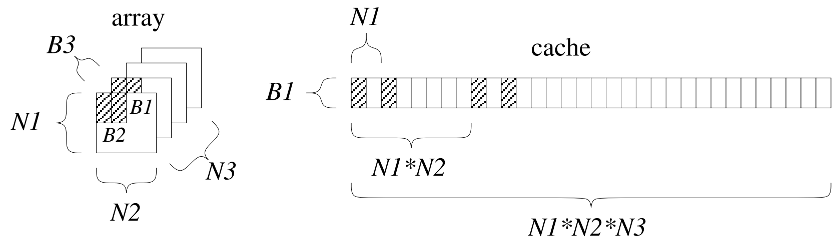
2.2 Multidimensional Cache Partitioning

When not all the indices from the inner array dimensions are simultaneously present in the cache, the data of a partition is no longer contiguous in both memory and cache. For example, in Figure 4(a), a block of data with dimensions $B_1 \times B_2 \times B_3$ from a three-dimensional array is spread across the cache. The noncontiguous blocks forming a partition for one array must be carefully interleaved with the noncontiguous blocks forming the partitions for other arrays to avoid cache conflicts. Hence, the goal of multidimensional partitioning is to determine the starting addresses for these interleaved, noncontiguous partitions. These starting addresses are then used to derive the memory layout using the greedy algorithm of Figure 3.

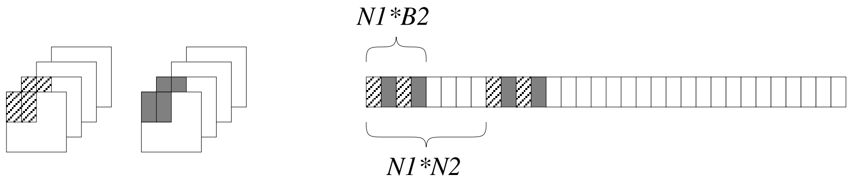
For arrays with dimensions $N_1 \times N_2 \times \cdots \times N_k$, the first task is to determine appropriate block dimensions $B_1 \times B_2 \times \cdots \times B_k$, where $B_i \leq N_i$, $1 \leq i \leq k$. To ensure that there is sufficient capacity in the cache for n_a partitions, the block dimensions must satisfy the capacity constraint $n_a \cdot B_1 \cdot B_2 \cdots B_k \leq c$, where n_a is the number of arrays and c is the cache size. Any set of dimensions $B_1, \ldots, B_k$ which satisfies this constraint is acceptable. A simple choice is a common block size $B_1 = \cdots = B_k = \lfloor \sqrt[k]{c/n_a} \rfloor$. In general, the data access patterns for the arrays in a loop nest may dictate a minimum block size in one or more dimensions. For example, the block size for the innermost dimension may set equal to a multiple of the cache line size. The block sizes for the remaining dimensions are then chosen subject to the above capacity constraint.

Once the block dimensions are known, the interleaving of the partitions is determined using a set of interleaving factors $n_1, \ldots, n_k$. These interleaving factors specify how successively larger groups of partitions can be interleaved without overlap. It is initially assumed that $N_1 \cdot N_2 \cdots N_k = c$ to derive the interleaving factors; this restriction is later relaxed. Figure 4(a) shows how $B_2 \cdot B_3$ subblocks, each of size B_1, are mapped into the cache starting at address 0. For a given index in array dimension 2, the starting addresses of two contiguous subblocks of size B_1 are separated by a distance N_1, as shown in Figure 4(a). In a space of size N_1, $n_1 = \lfloor N_1/B_1 \rfloor$ subblocks of size B_1 from different arrays may be placed. Hence, n_1 partitions for n_1 different arrays are interleaved to create a contiguous region of size $N_1 \cdot B_2$, as shown in Figure 4(b). To prevent these n_1 partitions from overlapping, their starting addresses in the cache must be separated (i.e., shifted in the cache) by a distance $g_1 = B_1$. If the first partition begins at address 0, the remaining starting addresses are $g_1, 2 \cdot g_1, \ldots, (n_1 - 1) \cdot g_1$. For example, in Figure 4(b), we have $B_1 = N_1/2$. Hence, the interleaving factor is $n_1 = 2$, and the starting addresses of the two partitions are 0 and g_1.

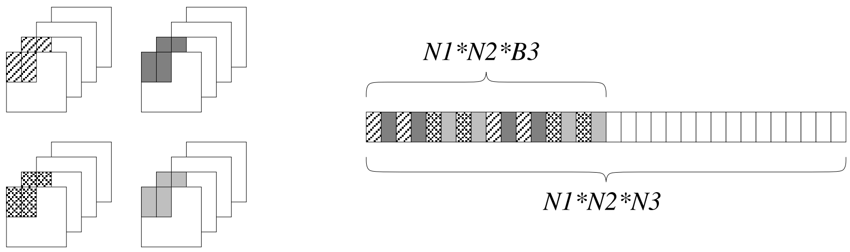
After interleaving a group of n_1 partitions in the cache, there are B_3 contiguous regions of



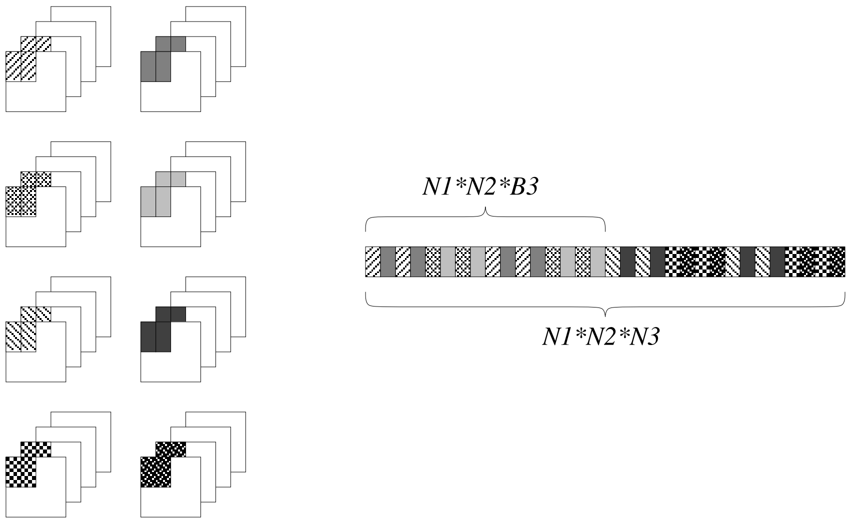
(a) Noncontiguous block of data in a three−dimensional array and its mapping in the cache



(b) First level of interleaving



(c) Second level of interleaving



(d) Final level of interleaving

Figure 4: Interleaving partitions in multidimensional cache partitioning

size $g_2 = N_1 \cdot B_2$ whose starting addresses are separated by a distance $N_1 \cdot N_2$, as illustrated in Figure 4(b). Identical groups of partitions of may be introduced into the space between these contiguous regions. The number of groups which can be interleaved within a distance of $N_1 \cdot N_2$ is given by the interleaving factor $n_2 = \lfloor (N_1 \cdot N_2)/(N_1 \cdot B_2) \rfloor = \lfloor N_2/B_2 \rfloor$. There are now a total of $n_1 \cdot n_2$ partitions. To ensure that the n_2 groups of n_1 partitions do not overlap, the groups must be separated or shifted by a distance g_2 in the cache. For example, in Figure 4(c), we have $B_2 = N_2/2$. Hence, the interleaving factor is $n_2 = 2$. The total number of partitions to this point is $n_1 \cdot n_2 = 4$, and the starting addresses are $0, g_1, g_2, g_2 + g_1$.

After interleaving n_2 groups of n_1 partitions in the cache, there is a contiguous region of size $g_3 = N_1 \cdot N_2 \cdot B_3$ in the cache, as illustrated in Figure 4(c). The cache size is $N_1 \cdot N_2 \cdot N_3 > g_3$, hence identical groups of partitions for other arrays may be introduced into the remaining space. The number of such groups is determined by the interleaving factor $n_3 = \lfloor (N_1 \cdot N_2 \cdot N_3)/(N_1 \cdot N_2 \cdot B_3) \rfloor = \lfloor N_3/B_3 \rfloor$. The total number of partitions is now $n_1 \cdot n_2 \cdot n_3$. To ensure that the n_3 groups do not overlap, they must be separated or shifted by a distance g_3 in the cache. For example, in Figure 4(c), we have $B_3 = N_3/2$. Hence, the interleaving factor is $n_3 = 2$. The total number of partitions to this point is $n_1 \cdot n_2 \cdot n_3 = 8$, and the starting addresses of the partitions are $0, g_1, g_2, g_2+g_1, g_3, g_3+g_1, g_3+g_2, g_3+g_2+g_1$.

In general, applying the above interleaving procedure for k-dimensional noncontiguous partitions yields

$$\begin{array}{llll} n_1 = \lfloor N_1/B_1 \rfloor, & n_2 = \lfloor N_2/B_2 \rfloor, & \ldots, & n_k = \lfloor N_k/B_k \rfloor, \\ g_1 = B_1, & g_2 = N_1 \cdot B_2, & \ldots, & g_k = N_1 \cdot N_2 \cdots N_{k-1} \cdot B_k, \end{array}$$

where n_i specifies the number of groups which can be interleaved at each point, and g_i specifies the separation between the groups to prevent overlapping. The base offset for a given group is $t_i \cdot g_i$, where $0 \le t_i < n_i$, and the starting address for each partition is determined by summing the group offsets across all dimensions, $\sum_{i=1}^{k} t_i \cdot g_i$.

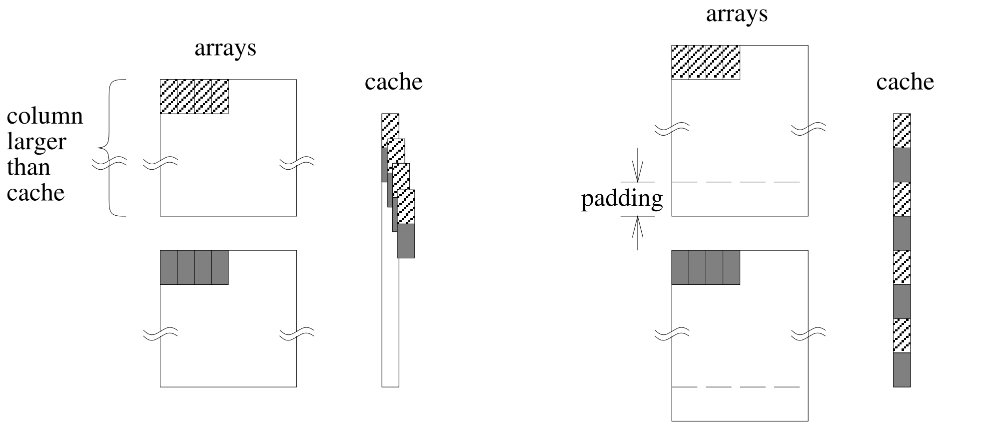
Upon completion of the interleaving procedure, it must be true that $n_1 \cdot n_2 \cdots n_k \ge n_a$. It is possible for this condition to be violated even if the capacity constraint is satisfied because of the truncation in the calculation of $n_1, \ldots, n_k$. In such cases, one or more of the block sizes $B_1, \ldots, B_k$ may be decreased in order to increase the corresponding interleaving factors by a sufficient amount to satisfy this condition.

The restriction $N_1 \cdot N_2 \cdots N_k = c$ is now removed, and the (unlikely) case of $N_1 \cdot N_2 \cdots N_k < c$ is now considered. In the preceding case of $N_1 \cdot N_2 \cdots N_k = c$, the final interleaving factor is given by $n_k = \lfloor (N_1 \cdot N_2 \cdots N_k)/(N_1 \cdot N_2 \cdots B_k) \rfloor = \lfloor N_k/B_k \rfloor$. In this case, the cache size is larger than $N_1 \cdot N_2 \cdots N_k$. To use the additional cache space for partitions, the final interleaving factor is computed as $n_k = \lfloor c/(N_1 \cdot N_2 \cdots B_k) \rfloor$. The preceding interleaving procedure is applied in the same way except for the change in computing n_k. Note that the block dimensions are still constrained by $n_a \cdot B_1 \cdot B_2 \cdots B_k \leq c$.

The final case to consider is $N_1 \cdot N_2 \cdots N_k > c$. Since the array size exceeds the cache size, *wraparound* occurs when mapping data into the cache. In this case, padding is introduced in the array dimensions which cause wraparound to ensure that a partition for a given array does not overlap with itself in the cache, and also to prevent partitions for different arrays from overlapping with each other. The innermost dimension i in which wraparound occurs, i.e. the smallest i such that $N_1 \cdot N_2 \cdots N_i > c$, is identified. In this dimension, the largest index m_i, $1 \leq m_i < N_i$, which does not cause wraparound is determined. In other words, the largest m_i such that $N_1 \cdot N_2 \cdots m_i \leq c$, but $N_1 \cdot N_2 \cdots (m_i + 1) > c$, is determined. The restriction $B_i \leq m_i$ is then introduced so that the contiguous subblocks of size $N_1 \cdot N_2 \cdots N_{i-1} \cdot B_i$ which result from interleaving do not exceed the cache size.

If wraparound occurs in dimension i, then wraparound will also occur in all remaining outer dimensions $i < j \leq k$ because the array size increases by a factor of N_j in each outer dimension j. Although it is possible to avoid the complications of wraparound for these outer dimensions by requiring $B_j = 1$, $i < j \leq k$, this approach is highly restrictive and may not satisfy other requirements on the block dimensions. To allow wraparound without overlap, an appropriate amount of padding is introduced in each outer dimension such that contiguous segments of data corresponding to adjacent indices in an outer dimension map to adjacent, nonoverlapping regions of the cache, as shown in Figure 5.

For each of the outer dimensions $i < j \leq k$ in which wraparound is permitted, the procedure for introducing padding is as follows. First, the size of the contiguous data block



(a) Overlapping of interleaved partitions due to wraparound in the cache

(b) Padding outer array dimension to avoid overlapping in the cache

Figure 5: The use of padding to handle wraparound in the cache

corresponding to a single index of dimension j is determined as $g_{j-1} = N_1 \cdot N_2 \cdots N_{i-1} \cdot B_i \cdots B_{j-1}$. We require $g_{j-1} \leq c$ to prevent the contiguous data region from overlapping with itself. Assuming that the start of this region maps to address 0 in the cache, the end of the region maps to address g_{j-1}. However, the next index in dimension j maps to address $c_{j-1} =$CACHEMAP$(N_1 \cdot N_2 \cdots N_{j-1})$ in the cache, assuming the array starting address maps to address 0. If $c_{j-1} \neq g_{j-1}$, then overlaps will occur due to wraparound for adjacent indices of dimension j. To prevent overlaps, a padding of p_{j-1} is required in dimension $j-1$ such that CACHEMAP$(N_1 \cdot N_2 \cdots N_{j-2} \cdot (N_{j-1} + p_{j-1})) = g_{j-1}$. The padding ensures that blocks of data corresponding to adjacent indices in dimension j map to adjacent, nonoverlapping regions of the cache. Because the blocks are adjacent in the cache after padding (i.e., there is no space between these blocks), the interleaving factor for dimension $j-1$ is $n_{j-1} = 1$. For all subsequent uses of dimension $j-1$, N_{j-1} is replaced with $(N_{j-1} + p_{j-1})$. The block size in dimension j is B_j, hence the block of data for B_j adjacent indices occupies a contiguous region of size $g_j = N_1 \cdot N_2 \cdots N_{i-1} \cdot B_i \cdots B_{j-1} \cdot B_j$. The above procedure is then repeated for dimension $j+1$.

The use of padding in the manner described above forces the interleaving factor to be 1 for each outer dimension $i \leq j < k$. For the outermost dimension k, the interleaving factor is $n_k = \lfloor c/(N_1 \cdot N_2 \cdots N_{i-1} \cdot B_i \cdots B_k) \rfloor$. The inner dimensions $1 \leq j \leq i-1$ are unaffected by the padding, hence the interleaving factor is still determined as $n_j = \lfloor N_j/B_j \rfloor$. As before, it must be true that $n_1 \cdot n_2 \cdots n_k \geq n_a$ to ensure that a sufficient number of partitions are created. If not, one or more of the block dimensions $B_1, B_2, \ldots, B_k$ are reduced to permit increasing the interleaving factors to satisfy the condition.

2.3 Extension of Cache Partitioning to Multiple Loop Nests

A program may contain $n_\ell \geq 2$ loop nests referencing a common set of arrays. The number of partitions required for each loop nest is determined by the number of arrays in the loop nest, and is generally different for each loop nest. Consequently, deriving the cache-partitioned memory layout for each loop nest individually results in different sets of starting addresses for the same arrays. To avoid conflicting requirements on starting addresses, we elect to use a single set of partitions for all loop nests, and to use the same array-to-partition assignments in all loop nests. Not all of the arrays are used in any one loop nest, hence the number of partitions in this set may be larger than required for a given loop nest. Furthermore, two arrays may be assigned to the same partition if they are not used in the same loop nest, hence there may be fewer partitions than arrays. Therefore, the extension of cache partitioning to multiple loop nests requires: (a) determining the number of partitions which satisfies all loop nests, and (b) assigning arrays to partitions when there are fewer partitions than arrays. The remainder of this section addresses these two aspects of the problem.

The problem of finding the required number of partitions for multiple loop nests is formulated as a graph-coloring problem. Let L denote a set of n_ℓ loop nests referencing a set A of n_a arrays. Let $U(\ell)$ denote the set of arrays referenced (read or written) in a loop nest $\ell \in L$. The number of partitions required individually by each loop nest ℓ is $|U(\ell)|$. A graph $G(V,E)$ is constructed with a set of vertices $V[G] = A$ representing the arrays, and a set of edges $E[G]$. If arrays a_1 and a_2 are both referenced in a loop nest, then there is an edge $e = (a_1, a_2) \in E[G]$. Consequently, the arrays referenced in a loop nest form a clique (a fully-connected subgraph) of size $|U(\ell)|$ in the graph $G(V,E)$. The goal is to label each vertex with a color such that no vertices connected by an edge have the same color, and the

```
GreedyMemoryLayout2(n_p,A,COLOR,C)::                          // A = set of arrays
                   // COLOR:A ↦ {0,1,...,n_p − 1} (output from graph coloring)
                          // C = {c_0,···,c_{n_p−1}} (starting addresses in cache)
 P = {0,1,...,n_p − 1}                                // unassigned partition indices
 q = q_0                                // q_0 =starting address of available storage
 foreach p ∈ P
   partition(p) = −1                         // initial partition mappings are undefined
 endfor
 do
   select a ∈ A                                              // selection is arbitrary
   ℓ = COLOR(a)                                                // get color for array
   if partition(ℓ) = −1 then                          // not yet assigned to a partition
     mapped_cache_address = CacheMap(q)
     foreach p ∈ P                                                   // determine gaps
       gap(p) = C(p)− mapped_cache_address
       if C(p) < mapped_cache_address then
         gap(p) = gap(p) + cache_size
       endif
     endfor
     select p_opt ∈ P where gap(p_opt) = min_{p∈P} gap(p)            // select minimum gap
     gap = gap(p_opt)
     P = P \ {p_opt}                                   // remove from available indices
     partition(ℓ) = p_opt                          // establish color-to-partition mapping
   else
     p = partition(ℓ)                               // color already assigned to partition
     mapped_cache_address = CacheMap(q)
     gap= C(p)−mapped_cache_address
     if C(p) <mapped_cache_address then
       gap = gap + cache_size
     endif
   endif
   START(a) = q + gap                                                     // insert gap
   q = START(a) + SIZE(a)                                // adjust start for next array
   A = A \ {a}
 while A ≠ ∅
```

Figure 6: Greedy memory layout algorithm for multiple loop nests

number of colors is minimized. The number of colors is then interpreted as the number of partitions n_p required to satisfy all loop nests, and vertices with the same color correspond to arrays which are assigned to the same partition.

Obtaining an optimal solution for graph coloring is an NP-complete problem [4]. As in any instance of a graph-coloring problem, the maximum number of colors is given by the number of vertices, $n_a = |A|$. However, for our formulation of the problem, the minimum number of colors is $n_r = \max_{\ell \in L} |U(\ell)|$ since there is at least one clique of n_r vertices in the graph, and no less than n_r colors are required to color this clique. This lower bound on the number of colors is known *a priori* by construction of the graph. If a solution with $n_p = n_r$ colors is obtained, then it is the optimal coloring of the graph.

The number of colors n_p obtained from graph coloring determines the required number of partitions. Cache partitioning is then used to obtain the starting addresses for a set of n_p partitions in the cache. The problem is to map colors in the graph to cache partitions and place the arrays in memory such that the sizes of the gaps inserted to enforce the partition mappings are minimized. The problem is constrained by the fact that identically-colored vertices in the graph represent arrays which share the same partition.

The greedy algorithm in Figure 6 is employed to reduce gap sizes using an approach similar to that used in Figure 3. The input consists of the result of graph coloring and the set of starting addresses for the partitions in the cache. The output is a mapping of colors to partitions and a memory layout for the arrays based on this mapping. The algorithm selects arrays in an arbitrary order for placement in memory. If the color assigned to the array has not yet been mapped to a partition, then one is chosen by computing gap sizes for all available partitions, then selecting the partition yielding the smallest gap. This selection implicitly determines the partition assignment for all remaining arrays sharing the same color. When one of these remaining arrays is later selected by the algorithm, the size of the gap inserted for the layout is computed using the previously-assigned partition since there is no longer any choice for that array. The complexity of the algorithm is $O(n_a \cdot n_p)$.

Determining the memory overhead from the greedy algorithm is complicated by the fact that $n_p \leq n_a$. A simple probabilistic approach can be employed to arrive at a reasonable estimate for the expected memory overhead. When the color assigned to an array has not yet been mapped to a partition, the distance to the closest available partition (i.e., the gap size) is assumed to be $(1/i) \cdot c$, where i is the number of unassigned partitions remaining and c is the cache size. However, when the color has already been mapped to a partition, the gap size is expected to be $c/2$. The expected overhead from combining these two cases is

$$o_m \approx \frac{\left(\dfrac{n_a - n_p}{2} + \displaystyle\sum_{i=1}^{n_p} \frac{1}{i}\right) \cdot c}{n_a \cdot d}, \tag{1}$$

where $d = N_1 \cdot N_2 \cdots N_k$ is the array size. The overhead diminishes rapidly as the data size increases relative to the cache size.

3 Analytical Performance Model

In Section 2, we described how cache partitioning removes conflicts entirely while maintaining high utilization of the available cache capacity. In this section, we derive a performance model to characterize the improvement which can be obtained from locality-enhancing loop transformations in the absence of conflicts.

Cache misses are incurred during the execution of a loop nest to load array data into the cache. If data is modified in the cache, it is written back from the cache to memory when it is replaced by new data. A typical loop nest may reference each array element one or more times, hence the execution of the loop nest *sweeps* across each array in memory one or more times. More specifically, at least one memory sweep is required per array to bring data into the cache, and one additional sweep of writebacks is required from the cache to memory for each array that is modified. The data traffic between memory and the cache during the execution of a loop nest can be characterized by the number of memory sweeps.

A loop transformation that enhances locality causes a reduction in the data traffic between memory and the cache, i.e., a reduction in the number of memory sweeps. We define the *sweep ratio* sr as

$$sr = \frac{s_b}{s_a}, \tag{2}$$

where s_b is the number of memory sweeps before applying a particular locality-enhancing loop transformation, and s_a is the number of sweeps after applying the transformation. This ratio provides an indication of the potential performance improvement which can be obtained; the larger the ratio, the greater the reduction in data traffic, and the greater the reduction in execution time.

The total execution time for a loop nest can be expressed as $T = T_c + T_m$, where T_c is the total time for computation and T_m is the total time for servicing cache misses and writebacks (i.e., accessing memory). A locality-enhancing loop transformation ideally reduces T_m without affecting T_c in order to reduce the total execution time. We assume that T_m is directly proportional to the amount of data traffic, and hence proportional to the number of memory sweeps. The fraction of the total execution time spent accessing memory is given by $f_m = T_m/(T_m + T_c)$. A compiler may estimate this fraction from the computation found in the body of the loop nest and knowledge of the cost of cache misses and writebacks.

The speedup S is given by the ratio of execution times before and after applying a transformation. With some algebraic manipulation, we obtain

$$S = \frac{T_c + T_m^b}{T_c + T_m^a} \cdot \frac{T_m^b/T_m^a}{T_m^b/T_m^a} = \frac{(T_c + T_m^b)/T_m^b}{(T_c + T_m^a)/T_m^a} \cdot \frac{T_m^b}{T_m^a} = \frac{1/f_m^b}{1/f_m^a} \cdot \frac{k \cdot s_b}{k \cdot s_a} = \frac{f_m^a}{f_m^b} sr,$$

where k is the constant of proportionality relating the number of memory sweeps and T_m. It is desirable to eliminate f_m^a to characterize the speedup as a function of the sweep ratio sr and the fraction f_m^b. The equations $f_m^a = T_m^a/(T_m^a + T_c)$ and $f_m^b = T_m^b/(T_m^b + T_c)$ are each solved for T_c and the resulting quantities are equated to each other. Solving the resulting equation for f_m^a and substituting this quantity into the above equation for speedup yields

$$S = \frac{sr}{f_m^b - sr \cdot (f_m^b - 1)}. \tag{3}$$

Equation 3 provides a means of gaining insight into limits on the performance gains of locality-enhancing transformations. The range of values for the sweep ratio is given by $sr \geq 1$ (assuming that applying a transformation reduces the number of memory sweeps) and the fraction f_m^b must be in the range $0 \leq f_m^b \leq 1$. If we assume that computation is infinitely fast, then $T_c = 0$ and $f_m^b = 1$. The speedup in this case is $S = sr/(1 - sr \cdot (1 - 1)) = sr$, which is governed entirely by the measure of data traffic in and out of the cache. The partial derivative of S with respect to f_m^b is positive for $sr > 1$, hence the speedup monotonically increases from $f_m^b = 0$ to $f_m^b = 1$. Consequently, the maximum occurs at $f_m^b = 1$ and the sweep ratio sr is the absolute upper bound on speedup.

More realistically, computation speed is finite and $f_m^b < 1$. As a result, the speedup will be less than the bound given by the sweep ratio. However, as the sweep ratio increases, there is also a bound determined by f_m^b. That is,

$$S_{max} = \lim_{sr \to \infty} \frac{sr}{f_m^b - sr \cdot (f_m^b - 1)} = \lim_{sr \to \infty} \frac{1}{f_m^b / sr - (f_m^b - 1)} = \frac{1}{1 - f_m^b}.$$

The partial derivative of S with respect to sr is also positive for $sr \geq 1$, hence the speedup asymptotically increases towards the limit of $1/(1 - f_m^b)$ as sr increases when $f_m^b < 1$.

4 Applications of Cache Partitioning

In this section, we apply the model developed in Section 3 to predict the performance improvements of two locality-enhancing transformations, loop fusion and tiling.

4.1 Loop fusion

Fusion of loops from adjacent loop nests combines their respective loop bodies into a single body and collapses their respective iteration spaces into one combined space [1]. In so doing, the number of iterations separating references to the same array is reduced, and array reuse can then be exploited to enhance cache locality. When loop nests are fused together, as illustrated in Figure 7, the resulting loop nest may reference a large number of arrays. Conflicts between these arrays lead to cache misses which negate the locality benefit of fusion. These conflicts occur not only when elements from different arrays are referenced for the first time (such as $a[i, j]$ and $b[i, j]$ in Figure 7), but also affect array elements which are reused in later iterations of the same loop nest (such as $b[i, j-1]$ and $b[i, j]$ in Figure 7). Cache partitioning maps data from different arrays to separate regions of the cache, ensuring that initial array references do not conflict. Cache partitioning also guarantees that array elements which are reused in later iterations of the fused loop nest do not conflict with other array data and remain in the cache for as long as necessary.

Equations 2 and 3 from Section 3 can be used to assess the improvement which results from fusing a sequence of two or more adjacent loop nests L into a single loop nest. Prior

```
do j=1,N
   do i=1,N
      b[i,j] = a[i,j] + a[i,j+1]
      c[i,j] = b[i,j] + b[i,j−1]
   end do
end do
  .
  .
  .
do j=1,N
   do i=1,N
      y[i,j] = c[i,j] + a[i,j]
      z[i,j] = y[i,j] + y[i−1,j]
   end do
end do
```

(a) Loop nests which reuse arrays

```
do j=1,N
   do i=1,N
      b[i,j] = a[i,j] + a[i,j+1]
      c[i,j] = b[i,j] + b[i,j−1]
        .
        .
        .
      y[i,j] = c[i,j] + a[i,j]
      z[i,j] = y[i,j] + y[i−1,j]
   end do
end do
```

(b) Fusion to exploit array reuse

Figure 7: Example of loop fusion

to fusion, each loop nest is executed completely before the next. Thus, a complete memory sweep is required for each time an array is referenced in a loop nest. Let $U(\ell)$ denote the set of arrays referenced (read or written) in each loop nest $\ell \in L$. The number of memory sweeps for each loop nest ℓ is then $|U(\ell)|$. Across all loop nests, the total number of sweeps for reading data into the cache is $s_b^r = \sum_{\ell \in L} |U(\ell)|$. Modified data in the cache must be written back to memory as it is replaced by incoming data in each loop nest. Thus, there is an additional *outgoing* sweep of writebacks in *each* loop nest modifying an array. Let $U_w(\ell)$ denote the set of arrays which are modified in a loop nest ℓ, where $U_w(\ell) \subseteq U(\ell)$. Across all loop nests, the number of additional sweeps due to writebacks is given by the total *number of times* arrays are modified in the loop nests. That is, $s_b^w = \sum_{\ell \in L} |U_w(\ell)|$.

In contrast, fusion combines the loop nests L into a single loop nest, ideally requiring only one incoming memory sweep for each array by reusing data from the cache. A cache-partitioned memory layout for the arrays in the resulting fused loop nest eliminates conflict misses which would otherwise occur due to an increase in the number of arrays referenced. Hence, the total number of incoming sweeps is $s_a^r = n_a$, where $n_a = \left|\bigcup_{\ell \in L} U(\ell)\right|$ is the total number of arrays in the fused loop nest. Clearly, $s_a^r \leq s_b^r$, since each array is referenced in at least one loop nest prior to fusion. Indeed, $s_a^r = n_a$ is the minimum number of sweeps that can be achieved for n_a arrays.

Writebacks still occur after fusion. However, an array which is modified in two or more of the the loop nests before fusion generates only one writeback sweep after fusion. Consequently, the number of additional writeback sweeps after fusion is given by the *number of arrays* which are modified in *any* of the original loop nests. That is, $s_a^w = \left|\bigcup_{\ell \in L} U_w(\ell)\right|$. Clearly, $s_a^w \leq s_b^w$, since multiple writeback sweeps are required when the same array is modified in different loop nests prior to fusion. Indeed, s_b^w is the minimum number of writeback sweeps that can be achieved, since $s_b^w \leq n_a$.

The sweep ratio for loop fusion is given by

$$sr = \frac{s_b^r + s_b^w}{s_a^r + s_a^w}.$$

Since $s_a^r \leq s_b^r$ and $s_a^w \leq s_b^w$, it is clear that $sr \geq 1$. A compiler may assess the profitability of fusion simply by computing this sweep ratio. If it is very close to one, then the locality benefit is minimal, and fusion may not be profitable. As the sweep ratio increases, the benefit from fusion increases, because fewer cache misses and writebacks are incurred. However, the improvement is ultimately limited by the bound of $S_{max} = 1/(1-f_m^b)$ determined in Section 3. An estimate for f_m^b which is small (e.g., $f_m^b < 0.1$) also indicates that fusion offers limited performance benefits, even if the sweep ratio is relatively large.

4.2 Tiling

Tiling is another important loop transformation for enhancing locality in the cache [10]. It is most effective in exploiting reuse carried by an outermost loop. In Figure 8(a), successive iterations of the t loop completely reuse the contents of arrays a, b, c, and d. However, the array data is unlikely to be found in the cache when it is reused in the outer t loop because it is replaced during the execution of the inner i, j loops. Tiling strip-mines the inner loops, then performs loop interchange to move the strip control loops outermost with respect to the t loop, as shown in Figure 8(b). This produces "tiles" of iterations with dimensions $BI \times BJ \times T$, where each tile references a sufficiently small portion from each array such that reusable data from all four arrays can remain in the cache. Data is reused from the

```
do t=1,T
  do j=1,N
    do i=1,N
      b[i,j] = a[i,j] + d[i,j]
      c[i,j] = b[i,j] + a[i,j]
      d[i,j] = d[i,j] + c[i,j]
    end do
  end do
end do
```

(a) Loop nest with outermost loop carrying reuse

```
do jj=1,N,BJ
  do ii=1,N,BI
    do t=1,T
      do j=jj,min(jj+BJ−1,N)
        do i=ii,min(ii+BI−1,N)
          b[i,j] = a[i,j] + d[i,j]
          c[i,j] = b[i,j] + a[i,j]
          d[i,j] = d[i,j] + c[i,j]
        end do
      end do
    end do
  end do
end do
```

(b) Tiling to exploit reuse carried by outermost loop

Figure 8: Example of tiling

cache during the execution of the t loop within each tile, dramatically reducing the number of cache misses.

However, the benefit of tiling is diminished if conflicts occur between the arrays referenced within a tile. In Figure 8, each of the four arrays contributes a portion of reusable data within a tile $BI \times BJ$. If the array portions map into overlapping regions of the cache, the resulting conflicts incur costly cache misses to reload data on *each* iteration of the t loop within a tile. The application of cache partitioning to the loop nest in Figure 8(b) maps data from each array into one of four partitions of size $BI \times BJ$ in the cache, which eliminates these conflicts for all T iterations of the t loop. Note that the block sizes for tiling are constrained by $4 \cdot BI \cdot BJ \leq c$, where c is the cache capacity, in order for the four partitions to fit in the cache without overlapping. Since two inner loops are tiled, the reusable data from each array is not contiguous. Hence, multidimensional cache partitioning is required.

Cache partitioning offers an additional advantage for tiling by simplifying the choice of the block size. The elimination of conflicts allows the selection of the largest possible block sizes. Previous research has resorted to reducing block sizes or array copy optimization in order to prevent the occurrence of conflicts [5]. Reducing the block size and copy optimization underutilize the cache capacity, and copy optimization may also introduce considerable execution overhead. Cache partitioning permits the use of a large block size without conflicts and does not require copying, thereby maximizing the locality benefit for tiling.

We use our model to predict the performance improvement obtained by tiling a single loop nest ℓ. Let $U(\ell)$ denote the set of arrays referenced in the loop nest ℓ, and $U_w(\ell)$ denote the subset of arrays which are modified. Prior to tiling, each execution of the outermost loop requires a complete memory sweep for each of the arrays in $U(\ell)$, and an additional writeback sweep is required for each of the arrays in $U_w(\ell)$. Hence, the total number of memory sweeps for the entire loop nest before tiling is given by $s_b = T \cdot (|U(\ell)| + |U_w(\ell)|)$, where T is the number of iterations of the outermost loop.

When the loop nest ℓ is tiled, reusable data is retained in the cache by executing tiles of iterations whose data can fit in the cache. We first consider tiling which does not require loop skewing. Each tile performs all T iterations of the original outermost loop. Cache misses are incurred at the start of each tile to bring the required data into the cache, then the same data is reused from the cache for the remainder of the tile. In addition, writebacks occur only when modified data in the cache is replaced by new data for the next tile. Consequently, the total number of memory sweeps with tiling is $s_a = |U(\ell)| + |U_w(\ell)|$. The sweep ratio for tiling is therefore $sr = s_b/s_a = T$. For large T, the resulting speedup after substitution of sr into Equation 3 rapidly approaches the limit of $S_{max} = 1/(1 - f_m^b)$.

However, loop skewing may be required to enable legal tiling in the presence of backward dependences carried by the outermost loop [11]. Skewing reduces the amount of reuse which can be exploited with tiling; it is not possible to perform all T iterations of the original outer loop entirely within a single tile. Later tiles reload the data referenced in earlier tiles into the cache, which increases the number of memory sweeps required.

If a single inner loop is skewed by a factor s, then tiled with a block size b_1, each tile performs only $\lfloor b_1/s \rfloor$ iterations of the outer loop. Thus, a total of $T/\lfloor b_1/s \rfloor$ adjacent tiles are needed to execute all T iterations of the outermost loop. Reloading the same array data in each additional tile effectively means additional memory sweeps for each array. Instead of a single sweep per array, $T/\lfloor b_1/s \rfloor$ sweeps are required, hence the sweep ratio for tiling in this case is given by $sr = T \cdot /(T/\lfloor b_1/s \rfloor) = \lfloor b_1/s \rfloor$. Since $\lfloor b_1/s \rfloor < T$, the resulting speedup will be lower than for tiling without skewing. Furthermore, the speedup is independent of T in this case. The compiler must select b_1 sufficiently large to make $sr > 1$.

If two inner loops are both skewed by a factor s, then both are tiled by b_2 to yield a block with dimensions $b_2 \times b_2$, a total of $2 \cdot (T/\lfloor b_2/s \rfloor) - 1$ sweeps are required to execute all T iterations of the outermost loop. The factor of 2 arises from skewing in two dimensions, which means that the same array data is reloaded in adjacent tiles along two dimensions. The sweep ratio is given by

$$sr = \frac{T}{2 \cdot (T/\lfloor b_2/s \rfloor) - 1} \approx \frac{\lfloor b_2/s \rfloor}{2}.$$

Although it appears that the sweep ratio is smaller when two inner loops are tiled instead of one, it must be noted that the block sizes are determined differently. When one inner loop is tiled, one-dimensional cache partitioning is used, hence the block size b_1 corresponds to the limit on the number of indices in the outermost dimension (B_k in Section 2.1), as determined by the size of the cache and the size of the contiguous data for the inner dimensions. On the other hand, tiling two inner loops implies the use of two-dimensional cache partitioning, which is not restricted by data contiguity. Hence, b_2 can be made larger than b_1 at the expense of interleaving noncontiguous partitions. Provided that $b_2 > 2 \cdot b_1$, tiling two inner loops can produce a higher sweep ratio than tiling a single inner loop for the same cache and array sizes. A compiler can use this information to determine the number of inner loops to tile. Note that these tradeoffs are relevant only when skewing is necessary; in the absence of skewing, reuse can be fully exploited by tiling a single inner loop.

4.3 Combining Fusion and Tiling

Finally, we assess the performance improvement obtained from the application of fusion followed by tiling when a sequence of loop nests L is surrounded by an outer loop which is executed T times. Fusion of the loop nests L yields a sweep ratio of $sr > 1$ if there is array reuse across the loop nests. The resulting loop nest from fusion is now surrounded by a loop which is executed T times. Tiling one or more of the inner loops to exploit the reuse carried by the outer loop can further increase the sweep ratio. The new sweep ratio is given by $sr' = sr \cdot T$ if no skewing is required, or $sr' = sr \cdot \lfloor b_1/s \rfloor$ if skewing is required to tile one inner loop. Note that in the presence of skewing, it is necessary for $\lfloor b_1/s \rfloor > 1$, otherwise $sr' < sr$, i.e., fusion alone provides better performance than combining fusion with tiling. Such a test may be incorporated into a compiler to assess the profitability of combining these transformations; a clear illustration of the usefulness of our model.

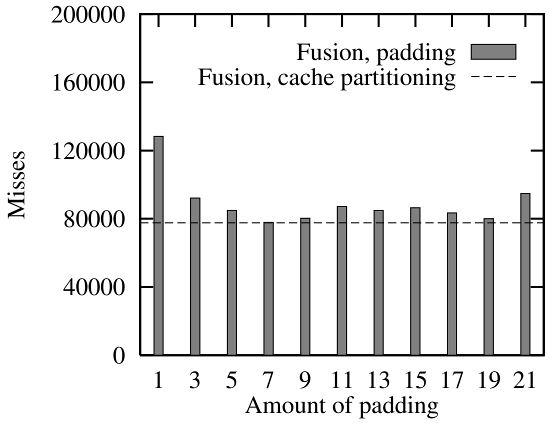
5 Experimental Results

In this section, we present experimental results that demonstrate the use of cache partitioning to maximize the benefit from locality-enhancing loop transformations, and the accuracy and usefulness of the performance model developed in Section 3. Our experiments are conducted on a single processor of the KSR1 multiprocessor from Kendall Square Research. The KSR1 processor includes a hardware performance monitor for measuring not only total execution time, but also the number of cache misses and the time spent servicing cache misses. The KSR1 processor implements a 64-bit superscalar architecture, and has a 256-KByte data *subcache*, which is 2-way set-associative with a random replacement policy. The subcache line size is 64 bytes, and the access latency is 2 processor cycles. A 32-Mbyte *local cache* is also associated with each processor to implement the cache-only memory architecture (COMA) of the KSR1 multiprocessor system. Misses in the subcache to access the local cache incur a latency of 20-24 processor cycles. In our experiments, data is placed in the local cache, and we assess the impact of misses in the processor data subcache. For brevity, all subsequent uses of the term *cache* denote the KSR1 data subcache.

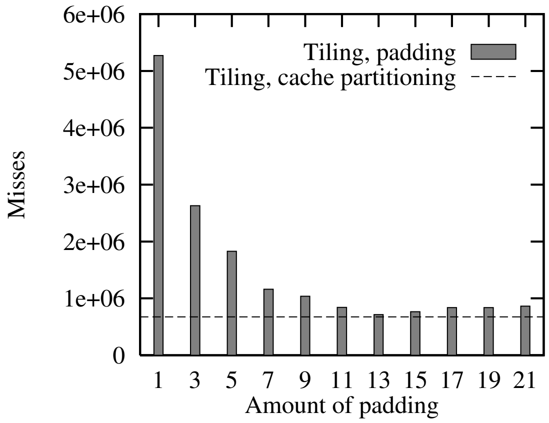
The codes used in our experiments consist of three kernels containing representative sequences of loop nests which are candidates for loop fusion and tiling. We use the Jacobi loop nest sequence used in the numerical solution of partial differential equations. From the Livermore Loops benchmark, we consider Kernel 18, an excerpt from a hydrodynamics application. Finally, from the application `qgbox` [8], which implements a finite-difference vorticity equation in ocean modelling, we consider a kernel of loop nests from the `calc` subroutine. Table 1 summarizes the characteristics of the code used in our experimentation,

Table 1: Characteristics of loop nest sequences

Loop nest sequence	Number of loop nests	Number of arrays	Array size	Sweep statistics s_b^r	s_b^w	s_a^r	s_a^w	Sweep ratio sr
Jacobi	2	2	500×500	4	2	2	2	1.50
`calc`	5	6	256×256	13	6	6	5	1.73
LL18	3	9	256×256	16	6	9	6	1.47



(a) Fusion



(b) Tiling

Figure 9: Effectiveness of cache partitioning for `LL18` on KSR

including the sweep statistics for loop fusion defined in Section 4.

The results are obtained from manually transforming the code and data for the loop nest sequences. To enable fusion and tiling, subroutine inlining and loop interchange are used where necessary to collect all loop nests together and ensure compatibility. Fusion-preventing dependences are overcome using the technique of shifting iteration spaces [7]. Legal tiling following fusion is enabled by loop skewing [11]. All code is instrumented to measure execution time and cache misses. The array sizes are large, and the entire data set cannot be contained in the cache. Cache-partitioned memory layouts are obtained using the techniques described in Section 2. The memory overhead for layouts did not exceed 5% for the array sizes given in Table 1.

The first set of results demonstrates the predictable cache behavior obtained with cache partitioning. Cache partitioning is compared with array padding [1]. For fusion and tiling of the LL18 loops, Figure 9 compares the number of misses obtained for various amounts of

Table 2: Experimental results for fusion—execution time

Loop nest sequence	Before fusion Exec. time (sec)	After fusion Exec. time (sec)	Speedup
Jacobi	0.286	0.214	1.34
`calc`	0.355	0.281	1.26
LL18	0.403	0.336	1.20

Table 3: Experimental results for fusion—cache misses

Loop nest	Before fusion		After fusion	
sequence	Meas. misses	Est. misses	Meas. misses	Est. misses
Jacobi	125062	125000	63243	62500
`calc`	109821	107330	49677	49152
LL18	136041	132098	77570	74305

array padding with the number of misses obtained from simply applying cache partitioning to the original arrays whose dimensions are powers of 2. Padding cannot guarantee the elimination of all conflicts. On the other hand, cache partitioning directly results in the minimum number of misses without the "guesswork" of padding. Consequently, the remainder of our results and comparisons are based on conflict-free cache-partitioned memory layout, and the improvements from locality enhancement reported below are *lower bounds* since they include the benefit of cache partitioning.

The next set of results shows the performance improvement from fusion, and how cache partitioning ensures that its full benefit is obtained. Table 2 presents the execution times for each loop nest sequence, before and after fusion, and the resulting speedup. Table 3 presents the measured number of cache misses and compares them to the estimated number of misses. Assuming that the bulk of the misses are due to the array references, the estimated number of cache misses is derived from the number of sweeps, the array size, and the cache line size. For example, `calc` requires $s_a^r = 6$ sweeps after fusion (see Table 1). The array size is 256×256, and the cache line size is 8 elements. Hence, the expected number of misses is $6 \cdot 256 \cdot 256/8 = 49152$. In Section 4, it was asserted that the number of cache misses after fusion is dictated by the number of sweeps s_a^r, and that this is the minimum which can be achieved. Our measured results agree closely with our estimates based on s_a^r, demonstrating

Table 4: Speedup from fusion

Loop nest sequence	Original miss time (sec)	Original exec. time (sec)	Miss fraction f_m^b	Sweep ratio sr	Speedup with fusion	
					est.	meas.
Jacobi	0.172	0.286	0.601	1.50	1.25	1.34
`calc`	0.143	0.355	0.403	1.73	1.21	1.26
LL18	0.175	0.403	0.434	1.47	1.16	1.20

Table 5: Experimental results for tiling after fusion

Loop nest sequence	Measured speedup	Tile size b_1	Skew factor s	Sweep ratio sr'	Estimated speedup
Jacobi	2.27	13	2	9	2.15
LL18 (128)	1.55	10	3	4.41	1.48
LL18 (256)	1.15	3	3	1.47	1.16

the effectiveness of cache partitioning in obtaining the full benefit of fusion by removing conflicts which would otherwise increase the number of misses and reduce performance.

The estimated speedup given by Equation 3 in Section 3 is also compared with the measured speedup for fusion. Table 4 provides the fraction of execution time spent servicing cache misses before fusion (as measured with the hardware performance monitor) and the estimated speedup computed with this fraction and the sweep ratio. Note that the sweep ratio accounts for writebacks as well as for cache misses. For all the experiments, the measured speedups agree closely with the analytical results, demonstrating how cache partitioning enables accurate performance modelling.

Finally, the application of tiling after fusion of the loop nest sequences is considered. The results are limited to Jacobi and LL18 because only these two kernels have surrounding outer loops carrying reuse which can be exploited by tiling. Loop skewing is required in both cases to enable legal tiling. Table 5 provides the speedups obtained from tiling with an outer loop of $T = 100$ iterations. The tile size and skew factors which yield the modified sweep ratio $sr' = sr \cdot \lfloor b_1/s \rfloor$, as discussed in Section 4, are also shown. For LL18, array sizes of 128×128 and 256×256 are considered to demonstrate how our model adjusts to changes in the model parameters. As before, the measured speedup is compared to the estimated speedup from Equation 3. The agreement of estimated and measured speedup in Table 5 demonstrates once again the importance of cache partitioning and the utility of our model.

If the upper bound on speedup given by $S_{max} = 1/(1 - f_m^b)$ is further considered, bounds of 2.51 for Jacobi and 1.76 for LL18 are obtained using the values of f_m^b in Table 4. The measured speedups in Table 5 are close to these bounds as a result of exploiting reuse with both fusion and tiling. Only the need for skewing limits further improvement. Without skewing, the sweep ratios for both Jacobi and LL18 would be greater than 10 (i.e., the product of the sweep ratios in Table 1 and $T = 100$). Consequently, the speedup in both cases would be pushed closer to the bound given by S_{max}.

In summary, our results demonstrate how cache partitioning maximizes the performance improvements from locality-enhancing transformations. The performance model developed in Section 3 has been validated by our results, and we have demonstrated its usefulness in providing insight into the measured performance. Given the current trend of increasing processor speeds relative to memory speeds, the potential improvements from enhancing locality with transformations such as fusion and tiling will be greater for future processors. Cache partitioning ensures that the full benefit is actually obtained.

6 Related Work

Array padding [1] is a data-reorganization technique which increases the size of the array dimension aligned with the storage order, and is most effective in reducing the occurrence of self-conflicts when the array dimensions are powers of 2. However, the amount of padding which minimizes the occurrence of conflicts between arrays is difficult to determine directly when data from different arrays must remain cached for reuse. Hence, arbitrary use of padding cannot guarantee the best performance.

Temam et al. [9] study conflicts arising from array references in loop nests typical of scientific applications. They analyze specific instances of self- and cross-conflicts, and suggest the use of padding and careful placement of arrays in memory. However, no detailed methodology is described for resolving conflicts.

Bacon et al. [2] 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. The method has been applied to one example resulting in a slight improvement in performance. Furthermore, their approach does not consider data reuse in outer loops, and cannot prevent cache conflicts for all reusable data.

Memory alignment [1] 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 small.

Lebeck and Wood [6] 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.

7 Concluding Remarks

This paper has made two specific contributions. First, it has presented a new technique for the elimination of cache conflicts among arrays in a loop nest. In this technique, called cache partitioning, the cache is logically divided into a number of nonoverlapping partitions, and the memory layout of arrays is adjusted to map different arrays into different partitions. Cache partitioning offers a number of advantages compared to previous approaches: (1) it consistently ensures that conflicts are eliminated, including those conflicts which diminish the benefit of exploiting data reuse carried by outer loops; (2) it avoids the trial-and-error needed to determine a padding factor or the necessary alignment; (3) it simplifies the choice of locality optimization parameters, such as the block size in tiling; (4) it enables accurate performance modelling, which permits a compiler to assess the profitability of locality-enhancing transformations. The technique requires no hardware support and incurs low memory space overhead in most applications. It does, however, require that arrays be allocated in a contiguous fashion in memory. Experimental results from a KSR system demonstrated the effectiveness and consistency of cache partitioning and its superiority over other techniques for reducing cache conflicts.

Second, this paper has presented a new model for assessing the benefit of locality-enhancing loop transformations in the absence of cache conflicts. The model is based on the sweep ratio metric, which can be determined by compiler analysis. Our results also validate our performance model, demonstrating its utility as a cost model for compiler optimization.

This work can be extended in a number of ways. The basic technique can be extended to deal with multiple levels of cache by incorporating additional constraints on array starting addresses to prevent conflicts in all cache levels. Consideration may also be given to caches with set associativity. Improved layout algorithms may be developed by paying closer attention to the array ordering to further reduce the memory overhead. Better estimates may be derived for the expected memory overhead. Cache partitioning can also be extended to derive the memory layout for parallel execution of loop nests for shared-memory multiprocessors with physically-distributed memory. In this case, additional constraints are required on the memory layout in each memory module to prevent conflicts between local and remote data in any processor's cache throughout the system.

References

[1] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. Technical Report UCB/CSD-93-781, Computer Science Division, University of California, Berkeley, 1993.

[2] 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.

[3] 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.

[4] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. W. H. Freeman, San Francisco, 1979.

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

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

[7] N. Manjikian and T. Abdelrahman. Fusion of Loops for Parallelism and Locality. Tech. Rep. CSRI-315, Computer Systems Research Institute, Univ. of Toronto, 1995.

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

[9] O. Temam, C. Fricker, and W. Jalby. Impact of cache interferences on usual numerical dense loop nests. *Proceedings of the IEEE*, 81(8):1103–1115, 1993.

[10] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proc. ACM Conf. on Prog. Lang. Design and Impl.*, pages 30–44, Toronto, Canada, 1991.

[11] M. E. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. on Parallel and Distributed Systems*, 2(4):452–471, 1991.