# Fusion of Loops for Parallelism and Locality

Naraig Manjikian and Tarek Abdelrahman

Technical Report CSRI-315
February 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.

# Fusion of Loops for Parallelism and Locality*

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

email: {nmanjiki,tsa}@eecg.toronto.edu

**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 reduce parallelism. In addition, performance losses result from cache conflicts in fused loops. We present new, systematic techniques which: (1) allow fusion of loop nests in the presence of fusion-preventing dependences, (2) allow parallel execution of fused loops with minimal synchronization, and (3) eliminate cache conflicts in fused loops. We evaluate our techniques on a 56-processor KSR2 multiprocessor, and show performance improvements of up to 20% for representative loop nest sequences. The results also indicate a performance tradeoff as more processors are used, suggesting a careful evaluation of the profitability of fusion.

**Index Terms — Loop fusion, Loop parallelization, Optimizing compilers, Data layout, Cache locality optimization**

*This research is supported by grants from the Natural Sciences and Engineering Council of Canada and from the Information Technology Research Center of Ontario. The use of the KSR2 was provided by the University of Michigan Center for Parallel Computing.

# 1 Introduction

The performance of data-parallel applications on cache-coherent shared-memory multiprocessors is significantly affected by data locality and by the cost of synchronization. Loop fusion is a code transformation which can be used to combine multiple parallel loops into a single loop, enhancing data locality and reducing synchronization. Fusion is not always legal in the presence of dependences between the loops being fused. Even when it is legal, fusion may introduce loop-carried dependences which reduce existing parallelism and introduce additional synchronization. Fusing a large number of loops also increases the number of arrays referenced in the fused loop, which gives rise to cache conflicts that degrade performance.

In this paper, we propose a new code transformation technique to cope with dependences that prevent fusion and parallelization, and propose a new data transformation technique which addresses the increased potential for cache conflicts. The loop transformation technique fuses loops, even in the presence of fusion-preventing dependences, by *shifting* iteration spaces prior to fusion. Parallel execution of a fused loop with minimal synchronization is enabled by *peeling* iterations to remove serializing dependences. We refer to this transformation as *shift-and-peel*. The data transformation technique adjusts the array layout in memory to eliminate mapping conflicts in the cache. We refer to this transformation as *cache partitioning*. The two transformations are integrated, and are applied in a systematic approach suitable for inclusion in a compiler.

The domain of the shift-and-peel transformation is program segments consisting of a sequence of loop nests that reuse a number of arrays, as shown in Figure 1. Each loop nest is assumed to be in a canonical form in which at least $k \geq 1$ outermost loops are fully parallel loops which carry no dependences. Dependences *between* any pair of loop nests are assumed to be uniform (a uniform dependence has a constant distance). In addition, there should not be any intervening code between the loop nests. We are interested in fusing the sequence of loop nests such that the resulting loop nest has $k$ outer loops that are fully parallel. We do not address the loss of parallelism in any of the original loop nests with more than $k$ parallel loops. This issue has been addressed in

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

Figure 1: Program model

previous work [5]. Our data transformation technique requires in addition that array accesses be *compatible*, i.e, accesses to a given array must have the same stride and direction across all loop nests. Although compatible accesses imply uniform dependences, the reverse is not necessarily true, hence compatibility is a stricter requirement. Nonetheless, compatible array accesses are typical in scientific applications.

Program transformations may be used to obtain a program segment that lies in the domain of our techniques. For example, interprocedural analysis can be used to identify candidate loops and to perform transformations such as loop extraction and loop embedding[1][8]. Loop distribution [3] may be used to produce parallel loop nests that adhere to the model. Since the loops will be eventually fused together, there is no loss of locality. Code motion [1] may be employed to obtain a sequence of parallel loops with no intervening code. Loop interchange and/or loop permutation [3] may be used to ensure compatible access patterns.

The remainder of this paper is organized as follows. Section 2 describes the shifting transformation for fusion. Section 3 describes the peeling transformation that enables the parallel execution

[1] Loop extraction moves loops from a procedure to the caller; loop embedding moves a loop containing a procedure call into the called procedure.

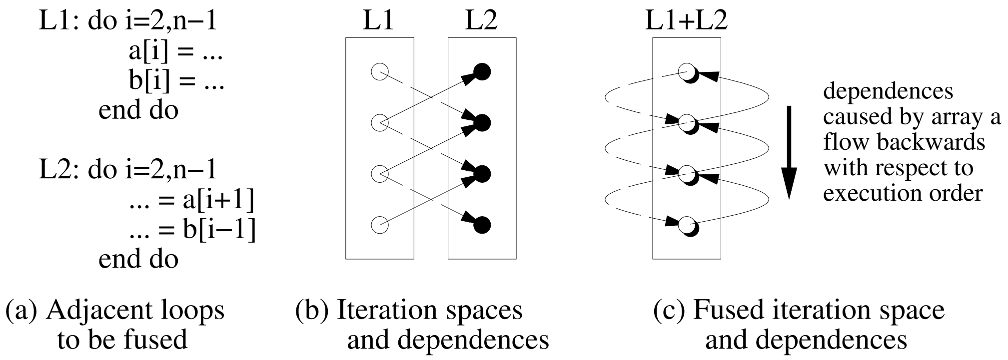



Figure 2: Illustration of illegal fusion of adjacent loops

of fused loops. In Section 4, cache partitioning is described. In Section 5, experimental evaluation of our techniques on a 56-processor KSR2 multiprocessor is presented. Section 6 describes related work. Finally, Section 7 presents concluding remarks and gives directions for future work.

## 2 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. In so doing, the number of iterations separating references to the same array is reduced, and array reuse can then be exploited to enhance register and cache locality. Register locality is enhanced through reuse of register-allocated array values in a single iteration of the fused iteration space [5]. Cache locality is enhanced through reuse of array elements across multiple iterations of the fused iteration space [3]. In either case, exploiting the reuse avoids costly references to main memory, thereby improving performance. In addition, fusion permits a reduction in the number of barrier synchronizations needed between parallel loops.

Fusion is not always legal. Array reuse between adjacent loops implies the existence of data dependences between different loops. These dependences are initially loop-independent since their source and sink iterations are in different iteration spaces. Fusion places the source and sink iterations of each dependence in the same iteration space. If the source and sink are different iterations of the fused space, the dependence becomes loop-carried. Fusion is legal only if no

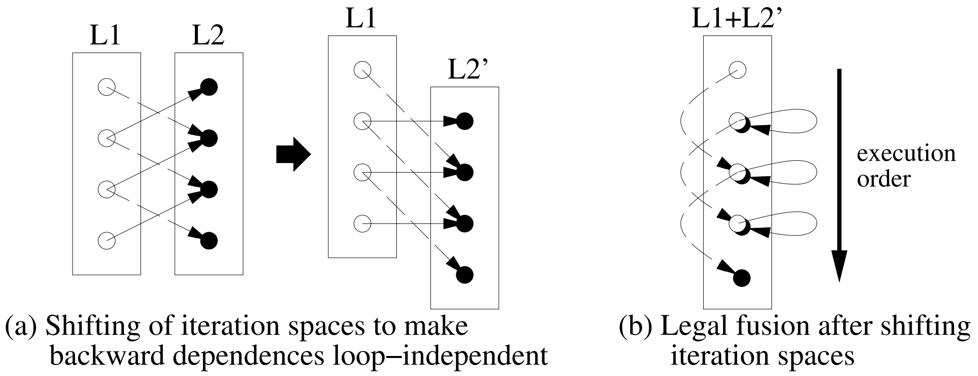



Figure 3: Shifting iteration spaces to permit legal fusion

resulting loop-carried dependence flows backwards with respect to the iteration execution order [11]. For example, both loops in Figure 2(a) reference arrays $a$ and $b$. This reuse gives rise to the loop-independent dependences between the iteration spaces shown in Figure 2(b). Fusion of the two loops combines the iteration spaces as shown in Figure 2(c). This fusion is not legal because the dependences caused by the reuse of $a$ become loop-carried after fusion and flow backwards with respect to the iteration execution order. The dependences caused by reuse of $b$ are also loop-carried, but flow forwards with respect to the execution and do not prevent fusion on their own.

We propose a simple technique to enable legal fusion in the presence of backward loop-carried dependences. The only necessary condition for this technique is for dependences to be uniform. The key 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 [4, 9]. 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. For example, the iteration space of the second loop in Figure 3(a) must be shifted by one iteration because of the backward dependence with a distance of one. Note that the shift increases the distance of the forward dependence. The loops may then be legally fused, as shown in Figure 3(b).

The resulting fused iteration space after shifting is not uniform. Some iterations contain com-

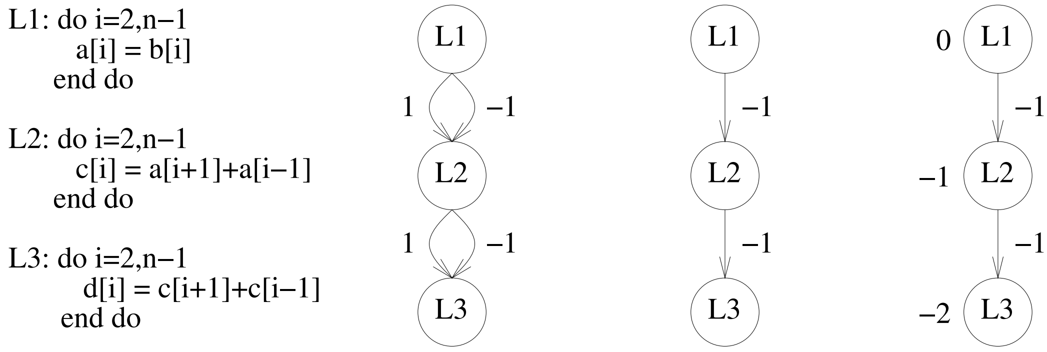
```
TraverseDependenceChainGraph(G)::
    foreach v ∈ V[G] do weight(v) = 0 endfor
    foreach v ∈ V[G] in topological order do
        foreach e = (v, v_c) ∈ E[G] do
            if weight(e) < 0 then
                weight(v_c) = min(weight(v_c), weight(v) + weight(e))
            endif
        endfor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

Figure 4: Algorithm for propagating shifts along dependence chains

putation from only the first loop, some contain computation from only the second loop, while the majority of the iterations contain computation from both loops. This nonuniformity must be addressed when implementing fusion with shifting.

In general, more than two loops may be considered for fusion, and fusion-preventing dependences may result from any pair of loops in the candidate set. Complex dependence relationships may exist between candidate loops in the form of *dependence chains* passing through iterations in different loops. These dependence chains are dictated by the array reuse and constitute iteration ordering requirements which must be preserved for correctness. If one loop is shifted, subsequent loops along all dependence chains passing through this loop must also be shifted to preserve the iteration ordering when the loops are fused. That is, shifts must be propagated along dependence chains. Consequently, it is advantageous to treat candidate loops collectively rather than incrementally one pair at a time.

We present a systematic method to determine the amount of shift needed for each iteration space to permit legal fusion of multiple loops. The technique is presented assuming one-dimensional loop nests for simplicity. However, it should be emphasized that it is applicable to multidimensional loop nests. An acyclic *dependence chain multigraph* is constructed to represent dependence chains. Each loop nest is represented by a vertex, and each dependence between a pair of loops is represented by an edge weighted by the dependence distance. 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 be multiple dependences between the same two loops. The multigraph is reduced to a simpler




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

Figure 5: Representing dependences to derive shifts for fusion

*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 loops. 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 directly by the original loop 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. The minimum value for the accumulated shifts is selected at each vertex to ensure that all backward dependences are removed. The algorithm is given in Figure 4. The complexity of the algorithm is linear in the size of the graph, and upon termination, the final vertex weights are interpreted as the amount by which to shift each loop *relative to the first loop* to enable legal fusion.

Figure 5 illustrates the above procedure for representing dependence chains and deriving shifts. The dependence chain multigraph for the loops in Figure 5(a) is shown in Figure 5(b). Figure 5(c) is the reduced dependence chain graph for the multigraph in Figure 5(b). Figure 5(d) illustrates the final shifts derived with the traversal algorithm on the graph of Figure 5(c).

Once the required shifts have been derived, the loops must be transformed appropriately to

```
do i=2,3
    a[i] = b[i]
end do
c[2]=a[3]+a[1]

do i=4,n−1
    a[i] = b[i]
    c[i−1] = a[i]+a[i−2]
    d[i−2] = c[i−1]+c[i−3]
end do

c[n−1] = a[n]+a[n−2]
do i=n−2,n−1
    d[i] = c[i+1]+c[i−1]
end do
```

(a) Direct method to implement shifts for fusion

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

c[n−1] = a[n]+a[n−2]
do i=n−2,n−1
   d[i] = c[i+1]+c[i−1]
end do
```

(b) Strip−mined method to implement shifts for fusion

Figure 6: Implementing shifts for fusion

complete the legal fusion. There are two methods to implement the shifts for fusion. In a direct approach, the loops are fused to interleave individual iterations from the original loops, and subscript expressions in statements from shifted loops are adjusted according to the shift. Shifting makes the resulting iteration space nonuniform, necessitating the introduction of guards in the fused loop. Iterations at both the start and end of the fused loop can be peeled out of the iteration space to avoid these guards, as shown in Figure 6(a). Nonetheless, guards may still be required if the original iteration spaces differ considerably.

A simple alternative to the direct approach is to strip-mine [3] the original loops by a factor of $s$, then fuse the resulting outer controlling loops to interleave iterations in groups of $s$, as shown in Figure 6(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. Peeling is required only for the end iterations in the original iteration spaces of shifted loops. Differing iteration spaces are easily accommodated by suitable modifications to the inner loop bounds. Note that with a strip size $s = 1$, each inner loop performs at most one iteration and effectively serves as a guard. 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.

Representation of the dependence chains requires determination of dependence distances. 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 [11]. The dependence distances between iterations from different loops can be easily determined from the subscript expressions for each array reference.

In this section, shifting has been described for one-dimensional loop nests. It should be emphasized that shifting is equally applicable when fusing more than one loop across multidimensional loop nests. The procedure described above is simply applied at each loop level, beginning with the outermost loop and working inward. The dependences carried at each level determine the dependence chains which are then used to derive the required shifts. Implementation of the shifts at each level is straightforward using the strip-mined approach described above.

## 3 Parallelization

Loop fusion enhances locality but may result in loop-carried dependences which prevent synchronization-free parallel execution of the fused loop. This is illustrated in Figure 7. The loops *L1* and *L2* in Figure 7(a) have no loop-carried dependences; the iterations of each loop may be executed in parallel without synchronization. Only a barrier synchronization is required to ensure that all iterations of *L1* have executed before any iterations of *L2*. However, when the iteration spaces of the two loops are fused as shown in Figure 7(b), loop-carried dependences result, requiring synchronization that serializes execution of blocks of iterations assigned to different processors.

We propose a simple method to remove serializing dependences which result from fusion. The central idea of our approach is to identify iterations which become the sinks of cross-processor dependences[2] in the fused loop, then *peel* these iterations from the iteration space. The only

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

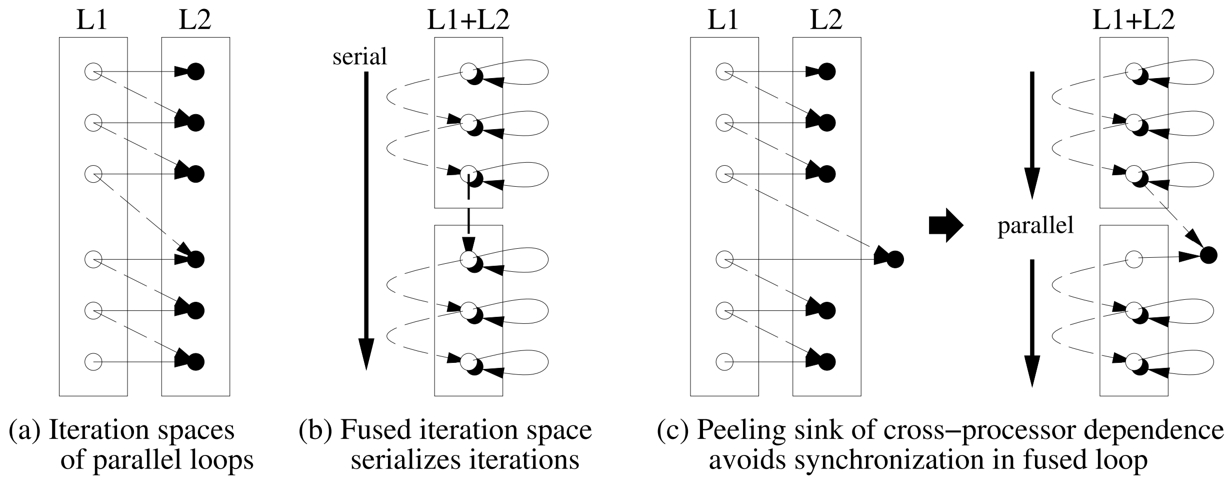
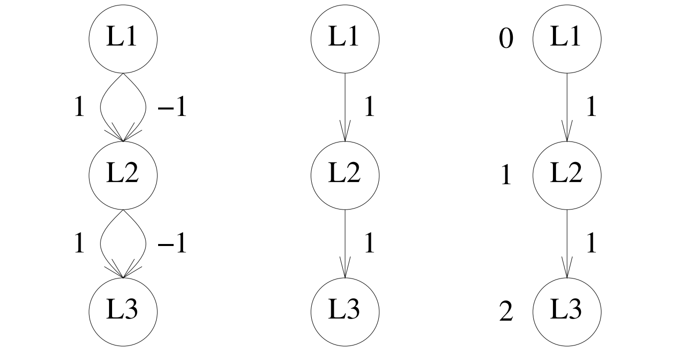



Figure 7: Fusion of parallel loops

necessary condition for this technique is uniform dependences, which force the peeled iterations to be located at block boundaries. The number of iterations which must be peeled is determined by the dependence distance. Peeling removes synchronization between blocks of iterations from the resulting fused loop, as shown in Figure 7(c). Note that 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 other iterations within each block have been executed.

In programs where more than two loops are fused, serializing dependences arise from one or more pairs of the original loops. Complex dependence relationships may exist between candidate loops in the form of dependence chains similar to those described in Section 2. The peeling of an iteration from one loop requires peeling of all subsequent iterations along all dependence chains passing through that iteration, i.e., peeling must also propagate along dependence chains.

We use a systematic method to directly determine the number of iterations to peel for each loop in the program. The technique is presented assuming one-dimensional loop nests for simplicity, but is applicable to multidimensional loop nests. Only the forward loop-carried dependences resulting from fusion need to be considered since the shifting removes backward loop-carried dependences. We use the same framework as in Section 2 for determining the shifts required to enable fusion. Forward dependences in the dependence chain multigraph are identified by edges with positive




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

```
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
```

(d) Transformed parallel code

Figure 8: Derivation and implementation of peeling

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 from the original multigraph and remains acyclic. The graph traversal algorithm described in Section 2 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. Upon termination, the interpretation of the final vertex weights is the number of iterations to peel relative to the first loop. Figure 8 illustrates this procedure using the dependence chain multigraph shown in Figure 5.

Implementation of peeling in the fused loop is straightforward using the strip-mined method described in Section 2. The loop bounds of the outer strip controlling loop represent a subset of the original iteration space because blocks of iterations are assigned to different processors for parallel execution. It is only necessary to modify the starting loop bound in the inner strip loops wherever peeling is required. For the example in Figure 8(d), the block of iterations is given by the range *istart...iend*. Based on results in Figure 8(c), the starting iteration from the second loop is peeled, as reflected by the starting bound of *istart+1*. Similarly, the two initial iterations are peeled from the third loop with the bound of *istart+2*. The other bounds, which are determined in part by the preceding derivation of shifts required for legal fusion, are unaffected.

The iterations peeled from the start of each block can only be executed after all preceding iterations along dependence chains 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 be executed in parallel without synchronization.

Iterations have also been peeled after shifting to enable fusion. However, these iterations are peeled from the end of blocks, rather than from the start as described above. Because there may be dependences between the iterations peeled from 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 opt to place peeled iterations from different processors in the same set such that all dependences are contained entirely within the same set. In this manner, synchronization-free parallel execution of sets of peeled iterations is still possible. For example, in the transformed code shown in Figure 8(d), peeled iterations are executed after a barrier to ensure all preceding iterations have been executed. The iterations executed after the barrier include those peeled from the end of block *istart...iend*, and also those peeled from the start of the block beginning at *iend+1*. It should be noted that there are minor differences in the transformed code for processors executing blocks at the boundaries of the full iteration space; for brevity, they are not shown here.

The traversals of the dependence chain graph to obtain the shifts required for fusion and the peeling required to permit synchronization-free parallel execution parallelization may both be performed simultaneously because they both employ the same basic algorithm. It is only necessary to maintain two values per loop vertex and perform two different computations at each vertex as the graph is traversed; the complexity remains linear in the size of the graph.

A final observation is that the approach to remove synchronization resulting from loop-carried dependences in parallel execution can also be applied to any single loop with forward loop-carried dependences. By identifying the source and sink statements for each dependence, then applying loop distribution to place statements in a separate loops, a set of loops is obtained which adheres to the model discussed in Section 1. The preceding approach can then be used to determine the peeling required to fuse the loops back together again such that parallel execution does not require synchronization. This approach compares favorably with that of Callahan [4] and Appelbe and Smith [2] in that it does not require expensive replication of code and in that it uses algorithms with lower complexity.

In this section, peeling has been described for one-dimensional loop nests. It should be emphasized that peeling is applicable for multidimensional loop nests. The procedure described above is simply applied at each loop level, beginning with the outermost loop and working inward. The number of iterations to peel at each level is derived from the chains of forward dependences carried at each level.

# 4 Cache Partitioning

Conflicts among elements of various arrays in the cache cause misses that can negate the locality benefits obtained by fusion. Conflicts occur when portions of data from different arrays map into overlapping regions of the cache, as illustrated in Figure 9(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 amount of data from each array which must remain cached in order to enhance locality, and hence an increase in the potential for conflicts.

A common solution to this problem is to *pad* array dimensions to perturb the mapping of data into the cache and reduce the occurrence of conflicts [3]. However, it is difficult to predict the amount of padding which minimizes the number of conflicts, particularly when the number of arrays is large. Figure 10 depicts the impact of various amounts of padding on the number of

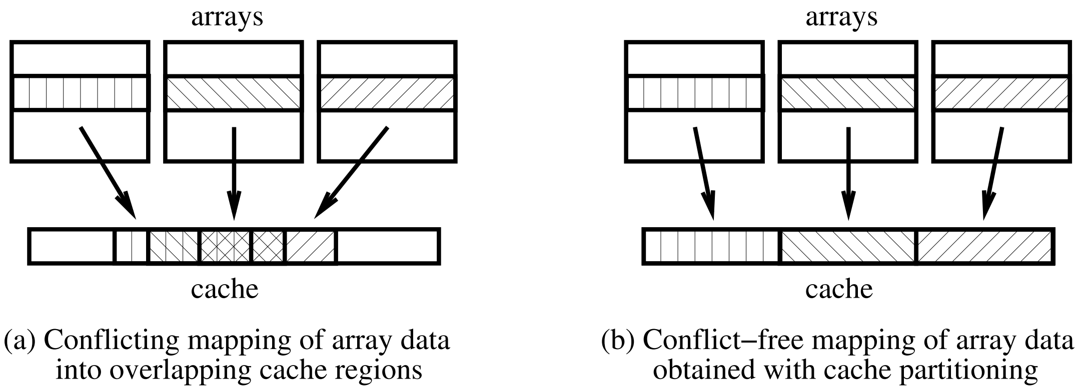



Figure 9: Avoiding conflicts in the cache

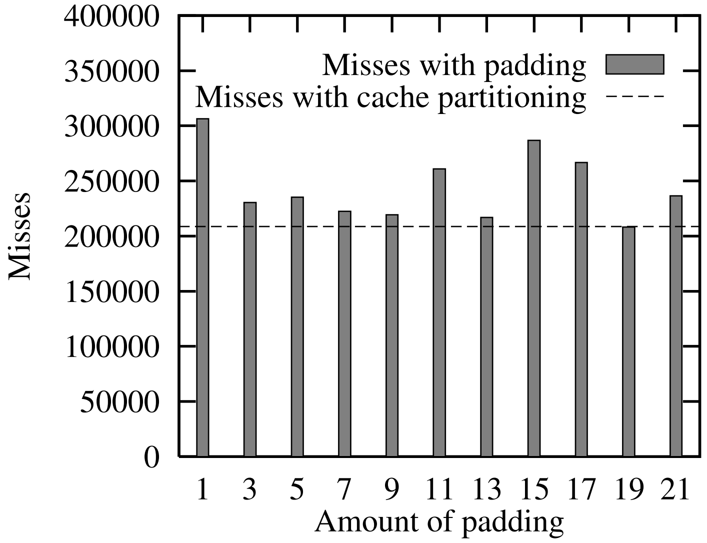



Figure 10: Cache misses from padding and cache partitioning

cache misses for the execution of a fused loop referencing nine arrays whose original dimensions are $512 \times 512$ (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 without the "guess-work" of padding. In this paper, we limit our presentation to *one-dimensional* partitioning; higher-dimensional partitioning is described in [6]. The basic idea behind cache partitioning is to logically

```
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 partiitons
      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 11: Greedy memory layout algorithm for cache partitioning

divide the cache into nonoverlapping partitions, one for each array, and to adjust the starting address of each array in memory such that each array maps into a different partition, as shown in Figure 9(b). The starting addresses of the arrays are adjusted by inserting appropriately-sized gaps between the arrays. Note that the logical cache partitions move through the cache during loop execution, but never overlap. Cache partitioning relies on compatible data access patterns such that 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. Evidence of the effectiveness of cache partitioning can be seen in Figure 10, which compares the number of misses from applying cache partitioning to the original arrays with the number of misses for various amounts of padding. Cache partitioning directly minimizes the number of misses and prevents erratic cache behavior.

The introduction of gaps between arrays is required to force each array to map into a separate partition of the cache. These gaps represent memory overhead which should be minimized. We employ the the greedy layout algorithm shown in Figure 11 to reduce the size of these gaps for a set of $n_a$ arrays. 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 CacheMap(). The complexity of the algorithm is $O(n_a^2)$, and the final layout guarantees a conflict-free mapping.

The partition size directly determines the maximum strip size for fusion using the strip-mined approach discussed in Section 2, where the largest possible strip size reduces the overhead of strip-mining. The strip size must be selected such that the total data referenced for each array in an inner loop 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 conflicts. 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 a predictable reduction in the number of misses, unlike the unpredictable outcome of padding.

## 5 Experimental Results

Results of experiments conducted on a Kendall Square Research KSR2 multiprocessor system using two representative loop nest sequences are presented in this section to illustrate the performance advantages that can be obtained from our techniques. From the Livermore Loops, Kernel 18 (`LL18`) is considered, which is an excerpt of three loop nests from a hydrodynamics code. These loop nests cannot be fused directly, as backward loop-carried dependences result. From the `qgbox` ocean modelling code [7], a sequence of five loop nests is considered from the `calc` subroutine. These loop nests also cannot be fused directly due to backward loop-carried dependences. In addition, differences between the iteration spaces prevent direct fusion; one of the loops has a larger iteration space than the other four. A problem size of $512 \times 512$ double-precision (8-byte) floating point values is used for both examples. `LL18` has 9 arrays, resulting in a total data size of 18 Mbytes. There are 6 arrays in `calc`, for a total data size of 12 Mbytes.

The outermost loops in each loop nest sequence are manually fused and parallelized using the techniques described in Sections 2 and 3. Fusion for `LL18` requires a shift of 1 for the second loop relative to the first loop, and a shift of 2 for the third loop relative to the first loop. Parallelization after fusion requires peeling one iteration from the third loop. In `calc`, two of the loops require shifting to enable fusion of all five loops; the largest shift is 2. The differences in the iteration spaces are accommodated by adjusting the min and max expressions within the inner loops after fusion. Parallelization requires peeling iterations from two of the original loops; the largest amount of peeling is two iterations.

The processors of the KSR2 are 2-way superscalar and operate at 40 MHz. Each processor has data and instruction *subcaches* of 256 Kbytes each. These caches are 2-way set-associative with a random replacement policy, and use a line size of 64 bytes. Each processor also contains 32 Mbytes of local memory to implement a Cache-Only Memory Architecture (COMA). Execution time and the number of cache misses are measured with the `pmon` performance monitoring hardware on each processor. The `pmon` hardware counts the number of data subcache misses, and this count is the one reported in the results given below. In the experiments described below, loops are executed repeatedly to permit reliable measurement of execution time, particularly for parallel execution.

The first set of results demonstrates the predictable cache behavior obtained with cache partitioning. Fused and unfused versions of the loops in `LL18` were executed on 16 processors. Figure 12 compares the number of misses obtained for various amounts of padding with the number of misses obtained from applying cache partitioning to the original arrays. These numbers reflect the misses

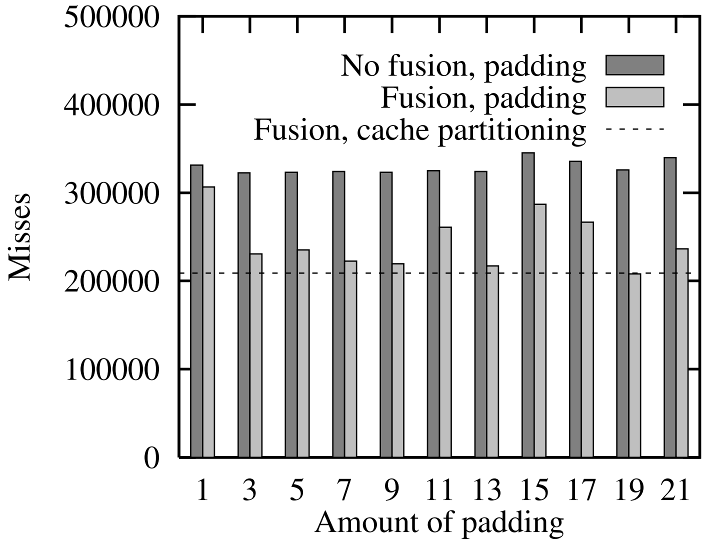
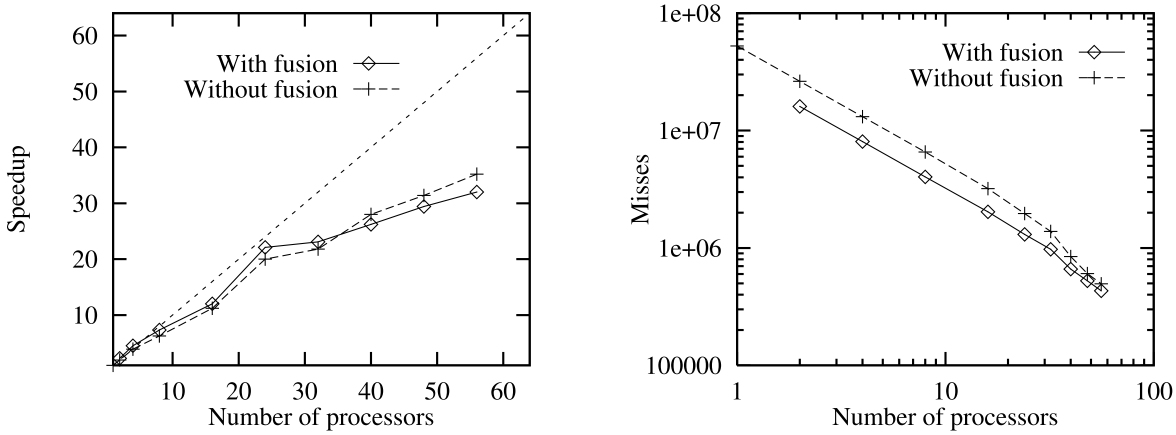



Figure 12: Effectiveness of cache partitioning for `LL18`

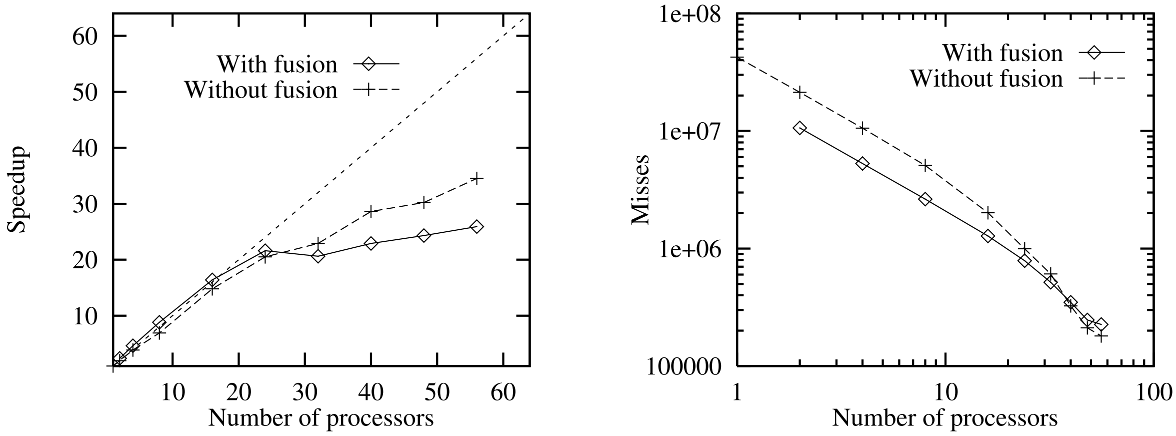
on one processor; the results are similar for the other processors in the same parallel run. The cache behavior with padding is erratic, as discussed previously in Section 4. On the other hand, cache partitioning directly results in the smallest number of misses. Results for 4 and 8 processors also yield the same conclusions. It is interesting to observe that the potential benefit of fusion is easily lost due to conflict misses; it is only when such misses are eliminated that the full benefit of fusion can be realized. The results indicate that padding cannot guarantee the avoidance of conflicts after fusion. This can be seen for certain values of padding for which the number of misses after fusion is comparable to the number of misses without fusion. Consequently, all our subsequent results and comparisons are based on cache-partitioned memory layout. The gaps introduced using this technique constitute an overhead of less than 2% for both `LL18` and `calc`, which is comparable to the overhead from padding.

The next set of results show the parallel performance of fused and and unfused versions of `LL18` and `calc` using cache-partitioned memory layout for up to 56 processors on the KSR2. For each parallel run (with or without fusion), the speedup is computed with respect to the unfused version of the code executing on a single processor (using cache-partitioned layout as mentioned above). The uniprocessor execution time is 78 seconds for `LL18`, and 64 seconds for `calc`. The number of




(a) (b)

Figure 13: Benefit of fusion for `LL18`




(a) (b)

Figure 14: Benefit of fusion for `calc`

cache misses is also measured for each run. Figure 13 shows the speedup curves and cache misses for `LL18`, and Figure 14 shows the same results for `calc`. 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 11% 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

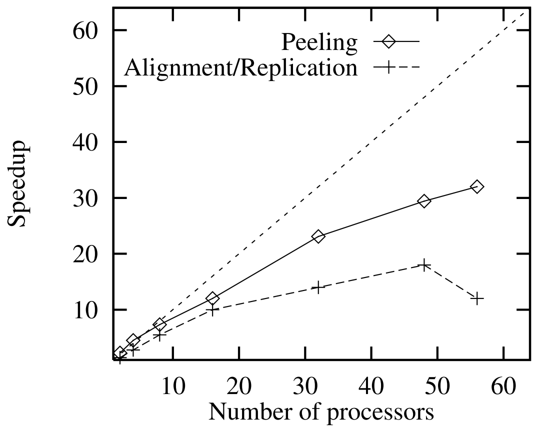



Figure 15: Performance of peeling and alignment/replication for `LL18`

such, they provide a lower bound on the performance improvement.

The results indicate a tradeoff when applying fusion and parallelization. Enhancing locality and reducing the number of barrier synchronizations with fusion must be weighed against the overhead of strip-mining used to implement shifting and peeling. For a given data size, using more processors increases the likelihood of data fitting in the cache of each processor, which reduces the relative gains from improving locality by fusion. The larger data size for `LL18` causes the benefit of fusion to overcome the overhead up to a larger number of processors than for `calc`. Furthermore, grouping peeled iterations from adjacent blocks in the manner described in Section 3 to permit them to be executed in parallel may increase the number of misses for nonlocal data. The results indicate that this effect becomes significant for a large number of processors, as can be seen for `calc`; the number of misses with fusion exceeds the number of misses without fusion beyond 40 processors. These observations suggest using knowledge of data sizes and cache sizes at compile-time to determine the profitability of fusion.

The final set of results compares our peeling transformation with the alignment and replication transformation proposed by Callahan [4] and Appelbe and Smith [2] (see Section 6). Figure 15 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 replication of code and data.

The results of this section have demonstrated the ability of our techniques to overcome the limitations of loop-carried dependences for fusion and parallelism. Although the performance improvement reported here for fusion is modest, it is expected that a larger improvement will result for faster processors. In addition, the relative cost of synchronization increases with faster processors. As a result, the benefit of reducing the number of barrier synchronizations, in conjunction with locality enhancement, will increase.

# 6 Related Work

In this section, we review related work and point out the unique aspects of our proposed techniques.

Warren [10] 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.

Callahan [4] proposes loop alignment to allow synchronization-free parallel execution of a loop nest, and uses code replication to resolve conflicts in alignment requirements. His approach potentially results in exponential growth in the number of statements in the loop, and can result in significant overhead because of the redundant execution of replicated code.

Porterfield [9] 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.

Appelbe and Smith [2] present a graph-based algorithm for deriving the required alignment, replication, and reordering to permit parallelization of an individual loop nest with forward direction loop-carried dependences. Their approach is similar to Callahan's approach in that it incurs overhead due to redundant execution of replicated code.

Kennedy and McKinley [5] use loop fusion and distribution to enhance locality and maximize parallelism. They focus on register reuse, and describe a fusion algorithm for avoiding the 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 shifting and peeling techniques described in this paper allow the systematic fusion and parallelization of multiple loops in the presence of loop-carried dependences. The adjustments needed for shifting and peeling are derived simultaneously for multiple loops using a simple graph-based framework. The techniques are simpler and more efficient than the alignment and replication techniques of both Callahan [4] and Appelbe and Smith [2]. The transformation algorithms for both shifting and peeling are linear in the number of loops being fused and can be easily automated in a compiler. A particularly unique aspect of this work is that it addresses cache conflicts that result from bringing references to many different arrays together in a single fused loop, an aspect which has not been adequately addressed in the past.

# 7 Summary and Conclusions

In this paper, we presented systematic techniques to improve data locality in data-parallel programs using fusion. We presented a shifting transformation to fuse loop nests, even in the presence of fusion-preventing dependences. The fusion of loop nests may result in loop-carried dependences, which prevent synchronization-free parallel execution. We presented a peeling transformation to overcome such dependences and allow parallel execution with minimal synchronization and without the overhead of code replication. Conflicts in the cache among array elements can negate the benefits of fusion. We presented cache partitioning as a method to eliminate such conflicts in a predictable manner.

The above techniques are evaluated using two real applications, and experimental results on a 56-processor KSR2 multiprocessor show up to 20% improvement in performance. Larger improvements can be expected as processor speeds continue to increase with respect to memory speeds. The results also indicate that performance tradeoffs exist. Performance gains resulting from fusion become smaller as the number of processors becomes larger, and the data used by each processor is more likely to remain cached. When the number of processors is sufficiently large, the overhead of fusion outweighs its benefits, and performance losses result. We conclude that the profitability of fusion should be evaluated with knowledge of the data size with respect to the cache size.

Future work includes the implementation of the techniques in an experimental compiler system being developed by the authors at the University of Toronto, and conducting a more comprehensive study of the profitability of fusion using a larger number of applications.

## References


[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, Massachusetts, 1986.

[2] W. Appelbe and K. Smith. Determining transformation sequences for loop parallelization. In *Proceedings of the 5th Workshop on Programming Languages and Compilers for Parallel Computing*, pages 208–222, 1992.

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

[4] Charles David Callahan. *A Global Approach to the Detection of Parallelism*. PhD thesis, Department of Computer Science, Rice University, March 1987.

[5] Ken Kennedy and Kathryn S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing—Sixth International Workshop*, pages 301–320. Springer-Verlag, Berlin, 1994.

[6] Naraig Manjikian and Tarek S. Abdelrahman. Cache-partitioned Tiling for Data Reuse Across Loop Nests. Tech. Rep. CSRI-307, Computer Systems Research Institute, University of Toronto, Toronto, Canada, October 1994.

[7] John D. McCalpin. Quasigeostrophic box model–revision 2.3. Technical report, College of Marine Studies, University of Delaware, July 1992.

[8] Kathryn S. McKinley. *Automatic and Interactive Parallelization*. PhD thesis, Department of Computer Science, Rice University, April 1992.

[9] Allan Kennedy Porterfield. *Software Methods for Improvement of Cache Performance on Supercomputer Applications*. PhD thesis, Department of Computer Science, Rice University, April 1989.

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

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