Fusion of Loops for Parallelism and Locality*

Naraig Manjikian and Tarek S. Abdelrahman
Department of Electrical and Computer Engineering
The University of Toronto
Toronto, Ontario, Canada M5S 1A4
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 improvements of up to 20% for representative loop nest sequences. The results also indicate a performance tradeoff as more processors are used, suggesting careful evaluation of the profitability of fusion.*

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 is 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 set of related loop and data transformations to address the above difficulties. 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 then enabled by *peeling* iterations to remove serializing dependences. We refer to this loop transformation as *shift-and-peel*. The data transformation technique adjusts the array layout in memory after fusion to eliminate mapping conflicts in the cache. We refer to this data transformation as *cache partitioning*.

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

Figure 1: Program model

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. A dependence *between* any pair of loop nests is assumed to be uniform, i.e., with a constant distance. In addition, there should be no 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 previous work [6]. 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 many scientific applications.

Program transformations may be used to obtain a program segment that lies in the domain of our techniques.

*This research is supported by grants from NSERC (Canada) and ITRC (Ontario). The use of the KSR2 was provided by the University of Michigan Center for Parallel Computing.

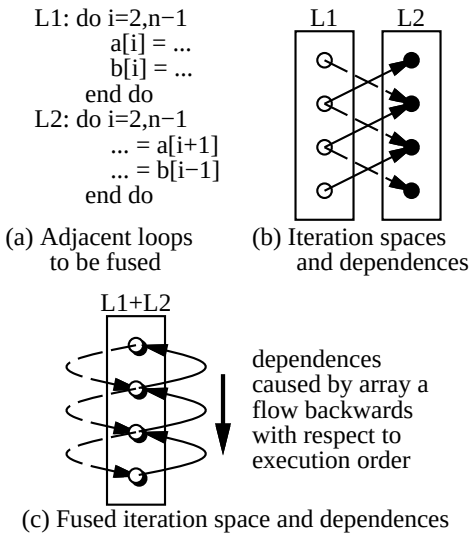


Figure 2: Illustration of illegal fusion of adjacent loops

Interprocedural analysis can identify candidate loops for fusion-enabling transformations such as loop extraction and loop embedding[9]. 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 and/or array dimension permutation [3, 7] may be used to ensure compatible access patterns.

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 [6]. 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. Fusion is legal only if it does not result in a loop-carried dependence that flows backwards with respect to the iteration execution order [10, 14]. For example, Figure 2 illustrates how the fusion of two loops may result in backward dependences.

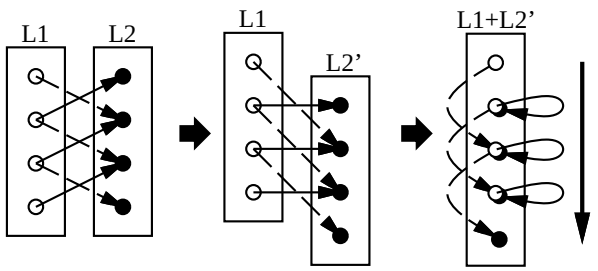


Figure 3: Shifting iteration spaces to permit legal fusion

We propose a simple technique to enable legal fusion of multiple loops in the presence of backward loop-carried dependences, based on the alignment techniques described in [4, 11]. The only necessary condition for this technique is uniform dependences. 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. The amount by which to shift is determined by the dependence distance. Other dependences between the loops are affected, but do not prevent fusion. For example, the iteration space of loop $L2$ in Figure 3 is 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 by one, but this does not prevent fusion.

In general, more than two loops may be considered for fusion, and fusion-preventing dependences may result from *any* pair of candidate loops. 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. Hence, 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 a directed edge weighted by the dependence distance. A forward depen-

```
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

dence 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 *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 the 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 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.

Once the required shifts have been derived, the loops must be transformed to complete the legal fusion. There are two methods to implement the shifts for fusion. In a direct approach, the loops are fused, and subscript expressions in statements from shifted loops are adjusted according to the shift. An advantage of this direct approach is the potential for register locality. However, shifting increases the reuse distance and hence limits reuse which may be exploited in registers. Furthermore, shifting makes the resulting iteration space nonuniform, requiring 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.

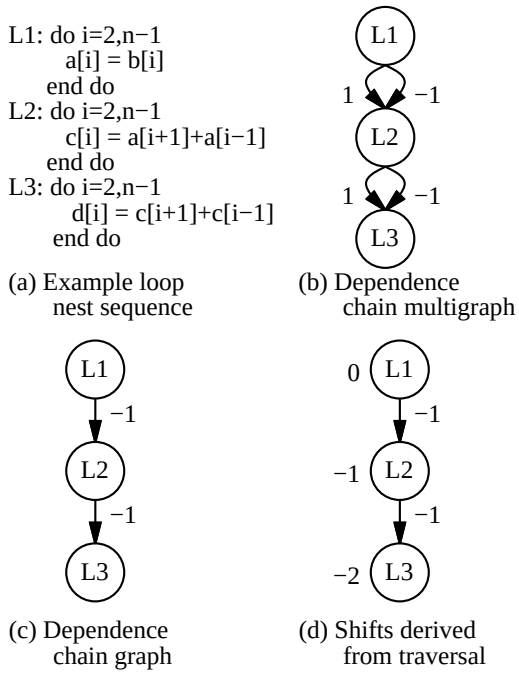


Figure 5: Representing dependences to derive shifts

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 in 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 later.

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. Although shifting an outer loop may change dependences with respect to inner loops, shifts are derived independently at each level using the original dependences

```
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

```
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

Figure 6: Implementing shifts for fusion

to remove backward dependences at all levels and permit subsequent parallelization in a uniform manner. Implementation of the shifts at each level is straightforward using the strip-mined approach described above.

3 Parallelizing Fused Loops

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, assuming static, blocked loop scheduling. The central idea of the method

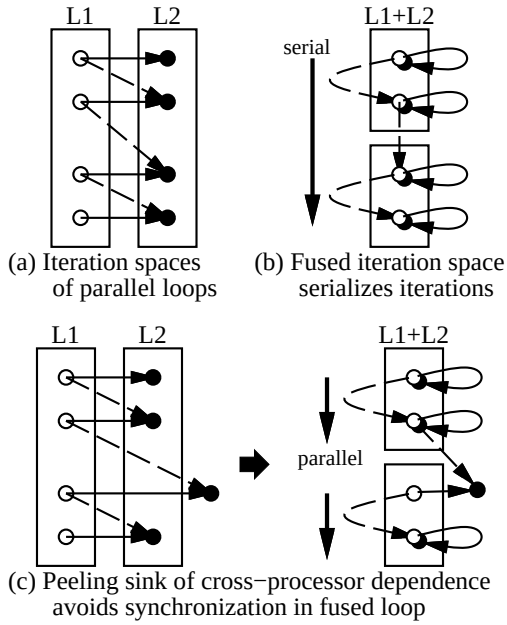
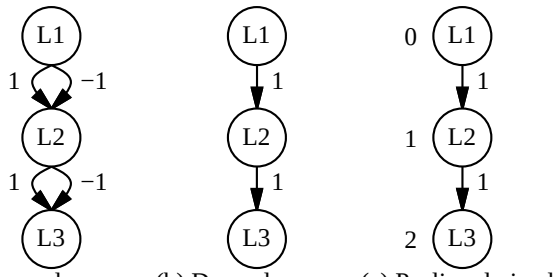


Figure 7: Parallel execution of fused loops

is to identify iterations which become the sinks of cross-processor dependences in the fused loop, and then *peel* these iterations from the iteration space. The only 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 after fusion, as shown in Figure 7(c). Loop-carried dependences still exist, but are contained entirely within a block executed by the same processor. The peeled iterations are executed only after all other iterations within each block have been executed.

In programs where more than two loops are fused, serializing dependences arise from *any* pair of the original loops. Complex dependence relationships may exist between candidate loops in the form of dependence chains similar to those considered when shifting iteration spaces. 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.

To determine the number of iterations to peel for each loop, we use the same graph-based framework used for shifting iteration spaces. However, only the forward dependences resulting from fusion need to be considered, since shifting removes backward dependences. In the dependence chain multigraph, such dependences are identified by edges with positive weights. The multigraph is reduced to a simpler graph by replacing multiple edges between two vertices with a single edge whose weight is the *maximum* from the original set of edges between the vertices



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

```
istart = START_ITER(my_pid)
iend = END_ITER(my_pid)
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

(as opposed to the minimum for shifting). When the maximum weight is positive, it determines the required number of iterations which must be peeled to remove serializing dependences. The graph traversal algorithm used to propagate shifts 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. All other edges are treated as having a weight of zero. Upon termination, vertex weights correspond to the number of iterations to peel relative to the first loop. Figure 8(a-c) illustrates this procedure using the dependence chain multigraph shown in Figure 5.

Implementation of peeling to parallelize the fused loop is straightforward using the previously-described strip-mined method. The loop bounds of the outer strip controlling loop determine the subset of the full iteration space assigned to each processor for parallel execution. Peeling is accomplished simply by adjusting the lower bounds in the inner strip loops where necessary. For the transformed code in Figure 8(d), which executes iterations *istart...iend*, one iteration from the second loop must be peeled, based on Figure 8(c), as well as two iterations from the third loop. The upper bounds, which resulted from shifting for legal fusion, are unaffected.

The iterations peeled from the start of each block of iterations may only be executed after all preceding iterations along dependence chains have been executed; a barrier synchronization is inserted to enforce this synchronization. Iterations peeled from the same block may have dependences between them. However, there are no dependences between sets of iterations peeled from different blocks because the dependences are uniform. As a result, these sets may be executed in parallel without synchronization following the barrier.

However, shifting for legal fusion also results in peeled iterations, and these iterations are always peeled from the end of a block. There may be dependences between the iterations peeled from the end of one block as a result of shifting and the iterations peeled from the start of an adjacent block after parallelization. We opt to place all adjacent peeled iterations in the same set. In this manner, dependences are contained entirely within each set, hence synchronization-free parallel execution of these sets is still possible. For example, in the transformed code shown in Figure 8(d), the peeled iterations include those peeled from the end of block *istart...iend* as a result of shifting, and also those peeled after parallelization from the start of the next block, beginning at *iend+1*. These iterations are executed in parallel with peeled iterations from other blocks following the barrier. The transformed code for blocks at the boundaries of the full iteration space is slightly different; for brevity, it is not shown here.

The combined shift-and-peel transformation is applied to a loop nest sequence assuming that the number of iterations per loop nest is greater than the number of processors. The number of peeled iterations is determined by the dependence distances. If this number exceeds the number of iterations per processor after loop scheduling, the transformation cannot be applied directly. In such cases, the loop nest sequence is divided to isolate the endpoints of the dependences with the greatest distance in separately-fused subsequences and reduce the number of peeled iterations.

A final observation is that peeling to remove serializing dependences can also be applied to any single loop with forward dependences. By identifying the source and sink statements for each dependence, then applying loop distribution to place statements in separate loops, a set of loops is obtained which adheres to our model. The shift-and-peel transformation can then be used to fuse the loops back together again such that serializing dependences are removed. 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

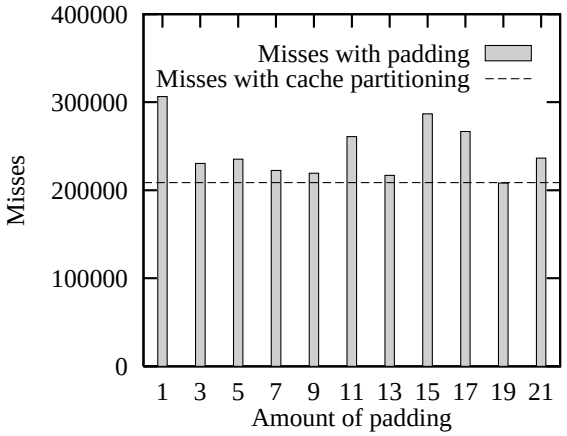


Figure 9: Misses from padding and cache partitioning

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 at each level. It is important to emphasize that both shifting and peeling are performed independently at each loop level using the original dependence information. In this manner, peeled iterations are grouped together in a systematic manner without requiring a more involved, case-specific dependence analysis, thus simplifying the transformation.

4 Cache Partitioning

Conflicts among elements of various arrays in the cache cause misses that can negate the locality benefits obtained by fusion, and can also reduce parallel performance by increasing memory contention. Conflicts occur when portions of data from different arrays map into overlapping regions of the cache. 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 9 depicts the impact of various amounts of padding on the number of cache misses for the execution of a fused loop referencing nine arrays whose original dimensions are 512×512 (the details of this experiment will be

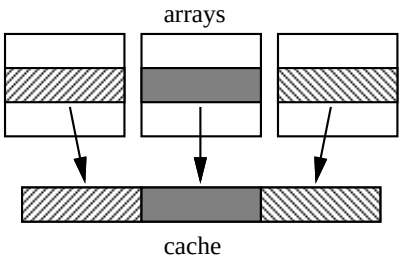


Figure 10: Avoiding conflicts in the cache

described in a later section). 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 the relatively large padding of 19.

We propose *cache partitioning* as a means of avoiding conflicts without the "guesswork" of padding. The basic idea behind cache partitioning is to logically 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 10. The starting addresses of the arrays are adjusted by inserting appropriately-sized gaps between the arrays. 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. Note that the logical cache partitions move through the cache during loop execution, but never overlap. Evidence of the effectiveness of cache partitioning can be seen in Figure 9, 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. In this paper, we limit our presentation to *one-dimensional* partitioning; higher-dimensional partitioning is described in [7]. In one-dimensional partitioning, the data in each partition is made contiguous by limiting the number of indices from only the outermost array dimension to make the data from an array fit in a partition.

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, assuming a direct-mapped cache with a typical address mapping function CACHEMAP(). 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

```
GREEDYMEMORYLAYOUT(A, c)::
  n_a = |A|    // A = set of arrays
  s_p = c/n_a    // partition size (c = cache size)
  P = {0, 1, ..., n_a − 1}    // available partitions
  q = q_0    // start of storage in memory
  do
    select a ∈ A    // selection is arbitrary
    mapped_address = CACHEMAP(q)
    foreach p ∈ P do    // determine gaps
      target_address(p) = p · s_p
      gap(p) = target_address(p) − mapped_address
      if target_address(p) < mapped_address then
        gap(p) = gap(p) + c    // "wraparound"
      endif
    endfor
    select p_opt ∈ P where gap(p_opt) = min_{p∈P} gap(p)
    P = P \ {p_opt}
    START(a) = q + gap(p_opt)    // insert gap
    q = START(a) + SIZE(a)    // adjust start
    A = A \ {a}
  while A ≠ ∅
```

Figure 11: Layout algorithm for cache partitioning

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 complexity of the algorithm is $O(n_a^2)$.

The partition size directly determines the maximum strip size for fusion in the previously-discussed strip-mined approach, 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 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 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 [5] using two representative loop nest sequences are presented 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 [8], 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 512×512 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 this paper. 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. 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. Differences in 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.

Our experiments focus on the impact of misses in the data cache on performance. Execution time and the number of cache misses are measured with the `pmon` performance monitoring hardware on each KSR processor. Loops are executed repeatedly to permit reliable measurement of execution time, particularly for parallel execution. In our experiments, cache partitioning is applied assuming a direct-mapped cache. The KSR data cache is 2-way set-associative, but maintains only one address tag for each set of 32 contiguous cache lines and employs a random replacement policy for these sets [12]. The associativity permits two distinct addresses to map to the same cache location without incurring conflict misses. However, random replacement ejects 32 cache lines at a time. Hence, the benefit of the associativity is limited, and we opt to consider the cache as direct-mapped.

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 on one processor; the results are similar for the other processors in the same parallel run. The cache behavior with padding is erratic and unpredictable. On the other hand, cache partitioning directly results in the smallest number of misses. Results for 4 and 8 processors also yield the same conclu-

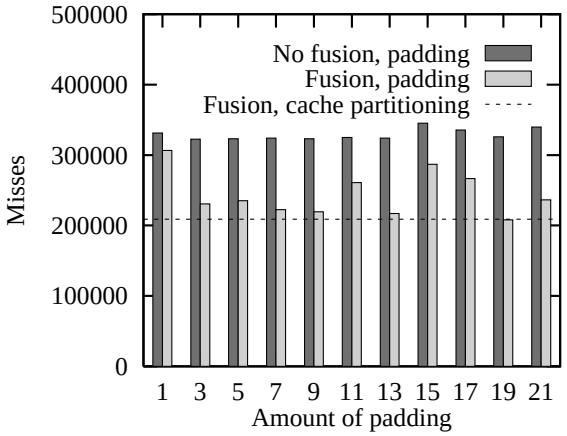


Figure 12: Effectiveness of cache partitioning for `LL18`

sions. 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 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 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 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

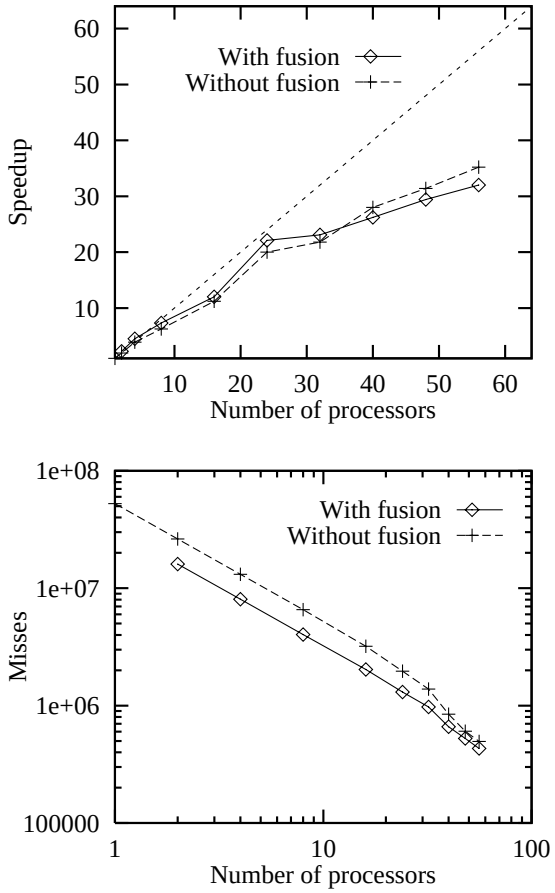
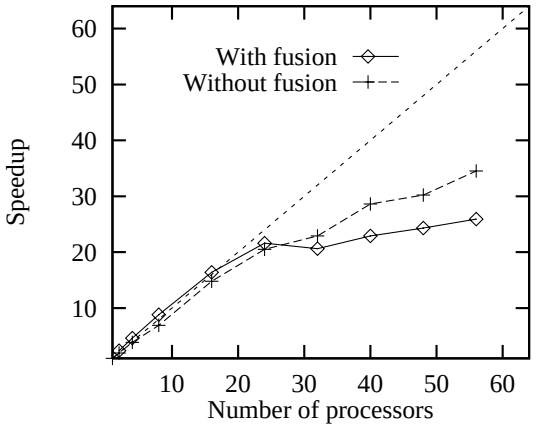


Figure 13: Benefit of fusion for `LL18`

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 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] (to be discussed in section on Related Work). 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



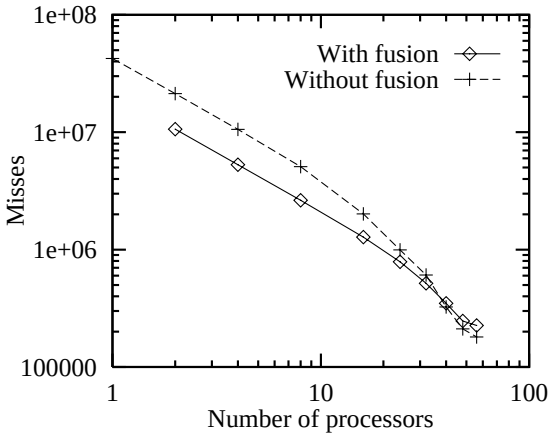


Figure 14: Benefit of fusion for `calc`

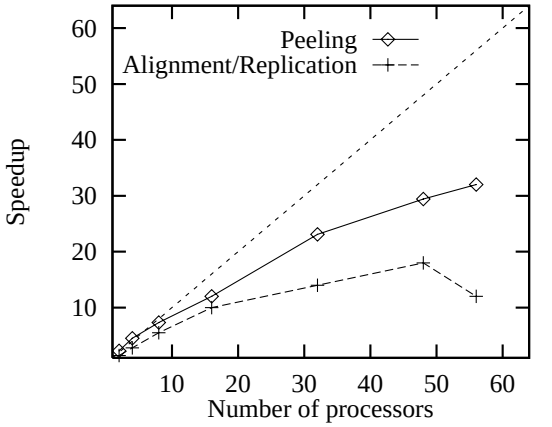


Figure 15: Alignment/replication for `LL18`

and two statements to enable synchronization-free parallel execution. The figure clearly indicates that superior performance is achieved using peeling, which 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 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

Warren [13] 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 with 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 [11] 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 dependences. Their approach is similar to Callahan's approach in that it incurs overhead due to redundant execution of replicated code.

Kennedy and McKinley [6] 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 fusion and parallelization of multiple loops in the presence of loop-carried dependences. The adjustments needed for shifting and peeling are derived 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 Concluding Remarks

In this paper, we presented new 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 have been 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.

Acknowledgements

We wish to thank the anonymous reviewers for their useful comments.

References

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

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

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

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

[5] Kendall Square Research. *KSR1 Principles of operation*. Waltham, Mass., 1991.

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

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

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

[9] K. S. McKinley. *Automatic and Interactive Parallelization*. PhD thesis, Dept. of Computer Science, Rice University, April 1992.

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

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

[12] R. H. Saavedra, R. S. Gaines, and M. J. Carlton. Micro benchmark analysis of the KSR1. In *Supercomputing '93*, pages 202–213, November 1993.

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

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