Array Data Layout for the Reduction of Cache Conflicts*

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—*The performance of applications on large-scale shared-memory multiprocessors depends to a large extent on cache behavior. Cache conflicts among array elements in loop nests degrade performance and reduce the effectiveness of locality-enhancing optimizations. In this paper, we describe a new technique for reducing cache conflict misses. The technique, called cache partitioning, logically divides cache capacity into equal parts, and allocates arrays in memory such that each array maps into a separate partition in the cache. We present experimental results from KSR and SGI machines to demonstrate the effectiveness of cache partitioning in eliminating cache conflicts and in realizing the full benefit of locality-enhancing techniques for both sequential and parallel execution.*

1 Introduction

In recent years, large-scale shared-memory multiprocessors have emerged as viable platforms for a variety of supercomputing applications; example systems include the Stanford Dash [10] and Flash [6], the Cray T3D [4], the Convex Exemplar SPP [3], the Kendall Square Research KSR1/2 [7], and the University of Toronto Hector [16] and NUMAchine [15]. These multiprocessors provide hardware support for shared-memory parallel programming, even though the memory is physically distributed throughout the system, as shown in Figure 1. The access latency for remote memory is greater than the latency for local or nearby memory. The effects of increased latency for remote memory are mitigated through the use of high-speed caches that reduce the frequency of accessing memory, and reduce contention arising from concurrent memory accesses during parallel execution. Parallel application performance depends in large part on locality in the cache.

Scientific applications often consist of loop nests that sweep through large data arrays whose size exceeds the capacity of a processor's cache, and hence incur a large number of memory accesses. Locality-enhancing transformations such as loop fusion [1] and tiling [8] can be used to exploit data reuse and reduce the number of memory accesses. However, cache conflicts among arrays in a loop nest eject reusable data from the cache, and the resulting conflict misses significantly reduce the effectiveness of locality enhancement [8]. The occurrence of conflicts is sensitive to cache hardware parameters, array sizes, and data access patterns in the loop nest. These factors make it difficult to guarantee that applying a given loop transformation will always improve performance.

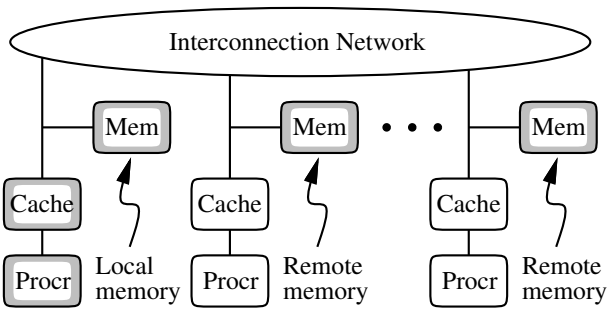


Figure 1: Shared-memory multiprocessor architecture

In this paper, we present a new technique called *cache partitioning* that eliminates conflicts within a loop nest by adjusting the memory layout of arrays to yield a conflict-free mapping of data into the cache. Cache partitioning logically divides the cache capacity into equal parts, and allocates arrays in memory such that each array maps into a separate partition in the cache. Cache partitioning is effective in maximizing the benefit of locality-enhancing loop transformations by ensuring that reusable data from different arrays is mapped into nonconflicting portions of the cache. In this manner, locality-enhancing transformations may be applied with the assurance that the potential benefit will not be diminished by the occurrence of cache conflicts.

The remainder of this paper is organized as follows. In Section 2, we introduce cache partitioning and present an algorithm for deriving a cache-partitioned memory layout for arrays referenced in a loop nest. In Section 3, we discuss the application of cache partitioning for two important locality-enhancing loop transformations, fusion and tiling. In Section 4, we present experimental results obtained from KSR

*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.

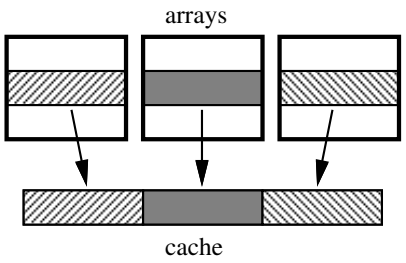


Figure 2: Avoiding conflicts in the cache

and SGI multiprocessors to demonstrate the importance and effectiveness of cache partitioning. In Section 5, we discuss related work. Finally, in Section 6, we conclude and give future research directions.

2 Cache Partitioning

Conflicts occur when data from different arrays maps into overlapping regions of the cache during the execution of a loop nest. Cache partitioning removes these conflicts by adjusting the memory layout of the arrays. The basic idea is to logically partition the cache into nonoverlapping regions, one for each array, and then adjust the starting addresses of the arrays in memory such that data from each array maps into a different partition, as in Figure 2(b). The starting addresses of the arrays are adjusted by inserting appropriately-sized gaps between the arrays in memory. Note that cache partitioning only affects the array starting addresses; no changes are required to the array index expressions.

Cache partitioning is particularly beneficial in avoiding the conflicts which eject reusable data from the cache. Data reuse which can be exploited in the cache is commonly associated with uniform dependences within a loop nest [5, 17]. A pair of references $A(f(\vec{\imath})), A(g(\vec{\imath}))$ to the same array A within a loop nest with iteration vector $\vec{\imath}$ generate uniform dependences if $f(\vec{\imath}) = h_A \cdot \vec{\imath} + c_f$ and $g(\vec{\imath}) = h_A \cdot \vec{\imath} + c_g$. The mapping $h_A : \mathbf{Z}^\ell \mapsto \mathbf{Z}^k$ is represented by a $k \times \ell$ matrix, where k is the array dimensionality and ℓ is the loop nest dimensionality, and c_f, c_g are constant vectors in $\mathbf{Z}^k$. In the presence of uniformly-generated dependences, spatial and temporal reuse of array data can be effectively exploited in the cache. In the absence of this uniformity, any potential reuse may be difficult to exploit [17].

In addition to the presence of uniform dependences, the benefit of cache partitioning is maximized when the data access patterns for the arrays referenced in a loop nest are *compatible*. Intuitively, array accesses with the same stride and direction are compatible. More formally, the references for a pair of arrays A and B are compatible if $h_A = h_B$. Compatibility ensures that once the mapping of array starting addresses into the cache is made conflict-free with cache partitioning, the mapping for the remaining data will also be

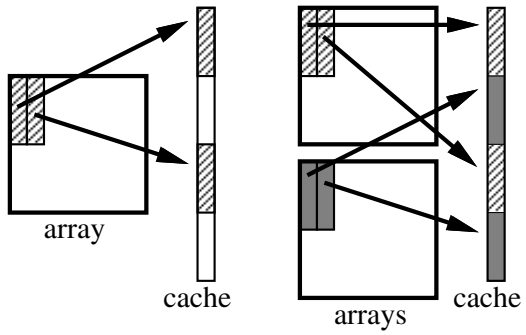


Figure 3: Multidimensional cache partitioning

conflict-free. During the execution of the loop nest, new data from each array is loaded into separate portions of the cache. The address mapping function causes new data from one array to replace older data from another array. Hence, the partition boundaries effectively move through cache. However, compatibility ensures that contents of the partitions never overlap.

Compatibility may seem to be a restrictive requirement, but it is normally satisfied in the presence of uniform dependences. If not, it is often possible to obtain compatibility from uniform dependences with appropriate data transformations. For example, if h_A and h_B are identical except for a permutation of rows, the array dimensions corresponding to those rows in one of the arrays may be permuted to obtain compatibility. If h_A and h_B differ in the stride for one dimension, array compression or expansion [9] along that dimension can be applied. If h_A and h_B differ in the sign in one dimension, the storage order in that dimension can be reversed for one of the arrays. In conjunction with code transformations (e.g., loop permutation), such data transformations are also necessary to improve the utilization of cache lines, i.e., exploit spatial reuse.

Because the cache capacity is limited, the partitions in the cache may not hold the entire contents of each array. Only a portion of each array may reside in the cache at any given time, or equivalently, a restriction is placed on the number of indices from one or more array dimensions. The simplest case is one-dimensional cache partitioning, where a restriction is placed on the number of indices from the outermost array dimension. All indices from the remaining inner dimensions for a given index of the outermost dimension are resident in a partition simultaneously. As a result, the data from each array is contiguous, and so are the cache partitions containing that data.

If the limit on the number of indices from the outermost dimension is too small to exploit the reuse of data in the outermost array dimension, it is necessary to perform multidimensional cache partitioning, which reduces the number of indices from other array dimensions and makes additional cache space available to accommodate a larger number of 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 4: Layout algorithm for cache partitioning

dices from the outermost dimension. In this case, the data in each partition is no longer contiguous because reducing the number of indices from inner dimensions skips over portions of the array in memory. Since the data is not contiguous in memory, the partitions containing this data in the cache are not contiguous either, as shown in Figure 3(a). These noncontiguous partitions must be carefully interleaved in the cache as shown in Figure 3(b) to ensure that they do not overlap and cause conflicts. For simplicity of presentation, the remainder of this section discusses one-dimensional cache partitioning. Multidimensional partitioning is described in [12].

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

Within the cache, the n_a partitions must be separated by a distance s_p to ensure that they do not overlap. If the first partition begins at address 0 in the cache, the starting addresses in the cache for the partitions are $0, s_p, 2 \cdot s_p, \ldots, (n_a - 1) \cdot s_p$. Each array must be assigned to a unique partition, hence the starting address of each array in memory must be adjusted such that the memory starting address maps to one of the valid partition starting addresses in the cache. Gaps between arrays in memory must be introduced 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 greedy layout algorithm shown in Figure 4 to reduce the size of these gaps when mapping arrays to partitions. The arrays are selected in an arbitrary order. A set of available partitions P is maintained, and each array to be placed in memory is assigned to a cache partition of size s_p which minimizes the distance between the starting address required for that partition and the end of the array most recently placed in memory. Although multiple memory addresses map into the selected partition, the address in free memory closest to the end of the most recently placed array is always used. Each partition selected in this manner is removed from the set of available partitions to ensure that two arrays are not assigned to the same partition. The algorithm assumes a direct-mapped cache with a typical address mapping function CACHEMAP(). The complexity of the algorithm is $O(n_a^2)$, and the final layout guarantees a conflict-free mapping. A straightforward analysis can show that the overhead from the gaps introduced by this algorithm (measured as the increase in the amount of memory required) is bounded by

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

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

3 Applications of Cache Partitioning

A loop transformation designed to enhance locality in the cache may produce a loop nest which references many arrays, and may also require that data from different arrays be retained in the cache across multiple loop iterations. A large number of arrays increases the potential for conflicts between arrays during the execution of the loop nest. These conflicts eject reusable data from the cache, incurring costly cache misses to reload the data into the cache. As a result, the performance benefit of the loop transformation is diminished. However, such conflicts can be eliminated with cache partitioning in order to maximize locality and yield consistently higher performance. In this section, we discuss the use of cache partitioning with two important locality-enhancing transformations, loop fusion and tiling.

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

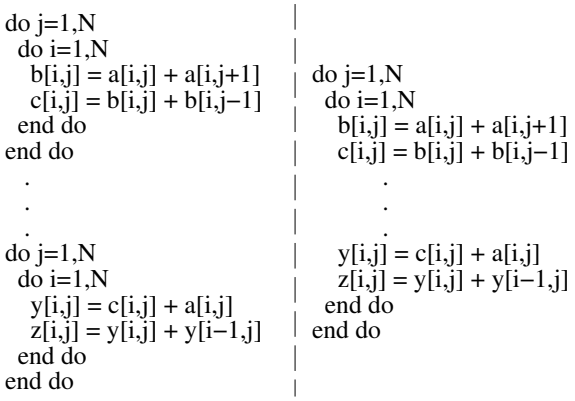


Figure 5: Example of loop fusion

3.1 Loop fusion

Fusion of loops from adjacent loop nests combines their respective loop bodies into a single body and collapses their respective iteration spaces into one combined space [1]. In so doing, the number of iterations separating references to the same array is reduced, and array reuse can then be exploited to enhance cache locality. When it is possible to fuse a number of loop nests together, as illustrated in Figure 5, the resulting loop nest may reference a large number of arrays. Conflicts between these arrays lead to cache misses which negate the locality benefit of fusion. These conflicts occur not only when elements from different arrays are referenced for the first time (such as $a[i,j]$ and $b[i,j]$ in Figure 5), but also affect array elements which are reused in later iterations of the same loop nest (such as $b[i,j-1]$ and $b[i,j]$ in Figure 5). Cache partitioning maps data from different arrays to separate regions of the cache, ensuring that initial references to array elements do not conflict. More importantly, cache partitioning guarantees that array elements which are reused in later iterations of the fused loop nest do not conflict with other array data and remain in the cache for later use.

The algorithm to apply cache partitioning with loop fusion is straightforward. Candidate loops for fusion are identified, and compatibility in the access patterns across the loop nests is enforced with appropriate code and data transformations. The loop nests are then fused, with any necessary adjustments to ensure that the fusion is legal [11]. Finally, the arrays referenced in the fused loop nest are identified, and the algorithm presented in Figure 4 is used to derive the memory layout for those arrays.

In some cases, it is not possible to fuse all candidate loop nests which use the same set of arrays into a single loop nest due to the presence of intervening code. As a result, there may be a sequence of two or more separately-fused sets of loop nests which share a common set of arrays. It still is possible to use cache partitioning to eliminate cache conflicts in each of the resulting fused loop nests. The first step is to determine an appropriate number of partitions to use throughout the entire sequence. This number may indeed exceed the maximum number of partitions needed in any of the loop nests because array-to-partition assignments are fixed throughout the sequence. A graph-coloring algorithm is then used to assign the arrays to the partitions such that the arrays used in each loop nest are assigned to different partitions. The complete algorithm to apply cache partitioning for multiple loop nests is given in [12].

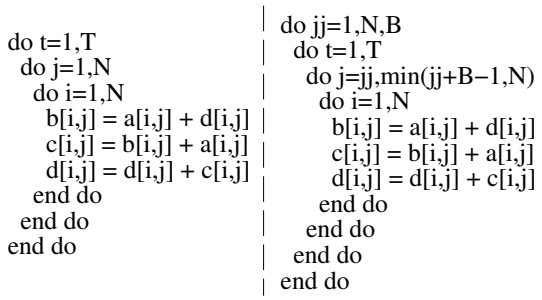


Figure 6: Example of tiling

3.2 Tiling

Tiling is another important loop transformation for enhancing locality in the cache [17]. Tiling is most effective in exploiting reuse carried by an outermost loop, as illustrated in Figure 6. In Figure 6(a), successive iterations of the t loop completely reuse the contents of arrays $a, b, c,$ and d. However, the array data is unlikely to be found in the cache when it is reused in the outer t loop because it is replaced during the execution of the inner i, j loops. The tiling transformation strip-mines an inner loop, then performs loop interchange to move the strip control loop outermost with respect to the t loop, as shown in Figure 6(b), which produces "tiles" of iterations with dimensions $T \times B \times N$. Each tile references a sufficiently small portion from each array such that reusable data from all four arrays can remain in the cache without being replaced. Data is reused from the cache during the execution of the t loop within each tile, dramatically reducing the number of cache misses.

However, the benefit of tiling is diminished if conflicts occur between the arrays referenced within a tile [8]. In Figure 6, each of the four arrays contributes a portion of reusable data of size $N \times B$ within each tile. If these array portions map into overlapping regions of the cache, the resulting conflicts incur costly cache misses to reload data on *each* iteration of the t loop within a tile. The application of cache partitioning to the loop nest in Figure 6(b) maps data from each array into one of four nonoverlapping partitions in the cache, eliminating conflicts for all T iterations of the t loop. Note that the choice of B is constrained by $4 \cdot B \cdot N \leq c$, where c is the cache capacity, in order for the four partitions to fit in the cache without overlapping.

The algorithm for applying cache partitioning with tiling

of a single inner loop is straightforward. Compatibility in the access patterns of the loop nest to be tiled is enforced with appropriate data transformations. The arrays referenced in the loop nest are identified, and the algorithm presented in Figure 4 is used to derive the memory layout for those arrays. The size s_p of each partition places an upper bound on the step size B for the inner loop to be tiled. For arrays with dimensions $N_1 \times N_2 \times \cdots \times N_k$, the number of indices from the outermost dimension of each array is limited to $B_k = \lfloor s_p/(N_1 \cdot N_2 \cdots N_{k-1}) \rfloor$. The step size B for the tiled loop should be such that no more than B_k indices of the outmost array dimension are referenced within a tile, i.e., $B \leq B_k$. The loop nest is then tiled with the size of B_k.

Cache partitioning offers an important advantage for tiling by simplifying the choice of the tile size B. The elimination of conflicts allows the selection of the largest possible size. Previous research has resorted to reducing the tile size or to copying in order to prevent the occurrence of conflicts [8]. Such approaches result in a significant underutilization of cache capacity or introduce execution overhead; both reduce the benefit of tiling. By permitting the use of a large tile size without conflicts and without resorting to copying, cache partitioning maximizes the locality benefit for tiling.

4 Experimental Results

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

The code used in our experiments consists of three kernels containing representative sequences of loop nests that are candidates for loop fusion and tiling. We use the Jacobi loop nest sequence used in the numerical solution of partial differential equations. From the Livermore Loops benchmark, we consider Kernel 18, an excerpt from a hydrodynamics application. Finally, from the application `qgbox` [13], which implements a finite-difference vorticity equation in ocean modelling, we consider a kernel of loop nests from the `calc` subroutine. Table 1 summarizes the characteristics of the code used in our experimentation.

[2]The 2-way set associativity permits up to two different data elements to map to the same cache location without conflicting, hence measured improvements from cache partitioning on the KSR represent a lower bound on the benefit of the technique.

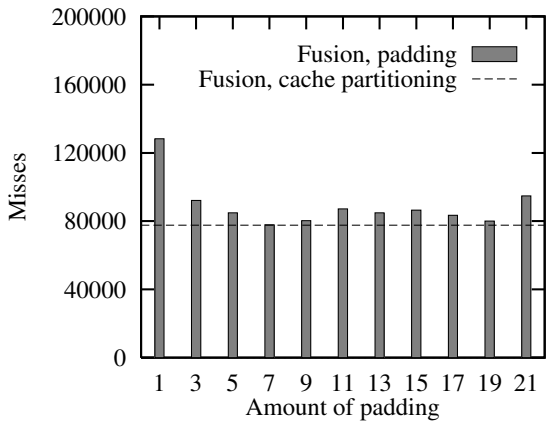
Table 1: Characteristics of loop nest sequences

Loop nest sequence	No. of loop nests	No. of arrays	Array size
Jacobi	2	2	500×500
`calc`	5	6	256×256
LL18	3	9	256×256

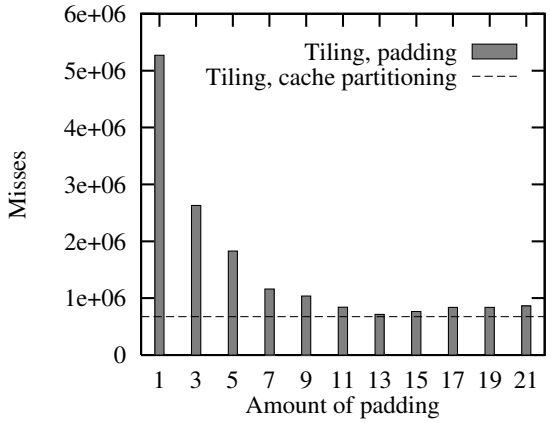
Our results are obtained by manually transforming the code and data for the loop nest sequences. To enable fusion and tiling, subroutine inlining and loop interchange are used where necessary to collect all loop nests together and ensure compatibility. Fusion-preventing dependences are overcome using the technique of shifting iteration spaces [11]. Legal tiling following fusion is enabled by loop skewing [18]. All code is instrumented to measure execution time and cache misses. The array sizes are large, and hence, the entire data set cannot be contained in the cache. Cache-partitioned memory layouts are obtained using the techniques described in Section 2. The memory overhead for layouts did not exceed 5% for the array sizes given in Table 1.

The first set of results demonstrates the predictable cache behavior obtained with cache partitioning. We compare cache partitioning with array padding [1], a well-known technique which attempts to reduce the occurrence of conflicts by increasing array sizes to alter the mapping of data into the cache. For the fused loops in LL18, Figure 7(a) compares the number of misses obtained for various amounts of array padding with the number of misses obtained from simply applying cache partitioning to the original arrays, whose dimensions are powers of 2. Figure 7(b) compares the number of misses when tiling the fused loop nests in LL18. In all our results, padding cannot guarantee the elimination of all conflicts. The number of misses may not be reduced significantly, and may actually increase, as in Figure 7(b). On the other hand, cache partitioning directly results in the best performance, realizing the full benefit of fusion and tiling without the "guesswork" of padding. Consequently, the remainder of our results and comparisons for the KSR are based on conflict-free cache-partitioned memory layout, and the improvements from locality enhancement reported below are *lower bounds* since they include the benefit of cache partitioning.

The next set of results demonstrates the performance improvement from fusion, and how cache partitioning ensures that its full benefit is obtained. Table 2 presents the execution times for each loop nest sequence, before and after fusion, and the resulting speedup. Table 3 presents the measured number of cache misses and compares them to the estimated



(a) Fusion



(b) Tiling

Figure 7: Cache partitioning for `LL18` on KSR

Table 2: Experimental results for fusion—execution time

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

number of misses. Assuming that the bulk of the misses are due to array references, the estimated number of cache misses is determined from the number of times arrays are expected to be loaded into the cache (assuming no conflicts), the array size, and the cache line. For example, fusion for `calc` results in a single loop nest which sweeps through the contents of 6 arrays. The array size is 256×256, and the cache line size is 8 elements. Hence, the expected number of misses in this case is $6 \cdot 256 \cdot 256/8 = 49152$. The agreement of the measured result with the estimate demonstrates the effectiveness of cache partitioning in obtaining the full benefit of fusion by avoiding conflicts which would otherwise increase the number of misses and reduce performance.

Table 3: Experimental results for fusion—cache misses

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

Table 4: Experimental results for tiling after fusion

Loop nest sequence	Meas. speedup	t (sec)	t_m (sec)	S_{max} $(\frac{t}{t-t_m})$
Jacobi	2.27	0.286	0.172	2.51
LL18 (128)	1.55	0.403	0.175	1.77

We now consider the application of tiling after fusion of the loop nest sequences. Our results are limited to Jacobi and LL18 because only these two kernels have surrounding outer loops carrying reuse which can be exploited by tiling. Loop skewing is required in both cases to enable legal tiling. Since we only consider one-dimensional cache partitioning in this paper, a smaller array size of 128×128 is used for LL18 to allow tiling only one inner loop, as in Figure 6. The larger array size of 256×256 requires tiling two inner loops in order to exploit the reuse, which in turn requires two-dimensional cache partitioning. Table 4 provides the measured speedup *after fusion and tiling* with an outer loop of $T = 100$ iterations, and compares the measured speedup with the absolute bound on speedup. The KSR hardware performance monitor isolates the time during which the processor is stalled accessing memory due to misses in the cache. Subtracting this memory access time t_m for the unfused and untiled execution from the total execution time t yields an estimate of the time for computation $t_c = t - t_m$. The absolute bound on speedup S_{max} is the ratio of the original execution time to the computation time, which is the execution time assuming that the tiling is successful in eliminating all memory accesses. In Table 4, the measured speedups are reasonably close to this bound. Conflicts are not the source of the discrepancy, since they are eliminated with cache partitioning in both cases (the results in Figure 7 confirm this statement). It is the need for loop skewing prior to tiling which limits the actual speedup. Without skewing, each array element is reused entirely within a single tile. However, skewing causes an array element to be reused in more than one tile, increasing the number of cache misses above the ideal minimum for tiling and reducing the measured speedup below the bound. Regardless of the skewing, cache partitioning is required to prevent conflicts between reusable portions of different arrays. Failure to avoid such conflicts can dramatically reduce performance, as indicated earlier in Figure 7.

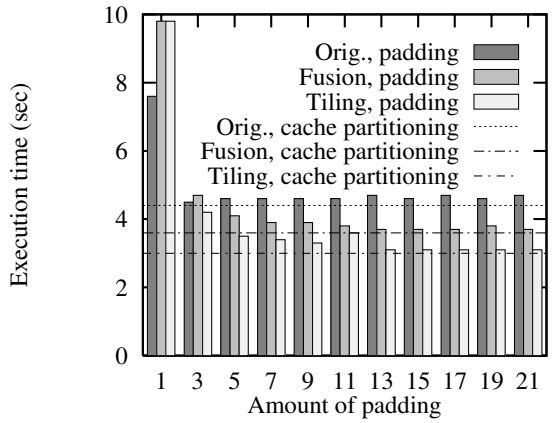


Figure 8: Cache partitioning for `LL18` on SGI Challenge

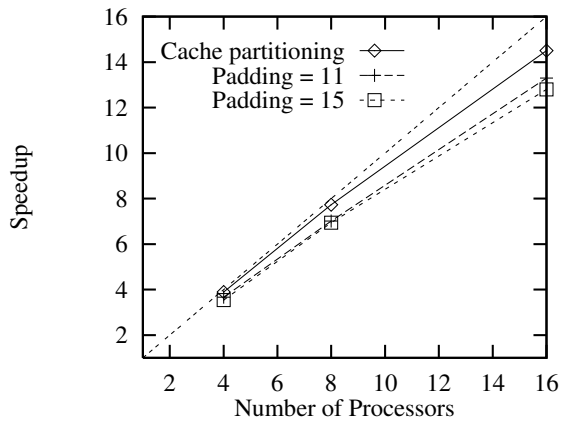


Figure 9: Speedup for fused `LL18` loop nests on KSR

As additional evidence for the effectiveness of cache partitioning on a different platform, we present results obtained from repeating the experiments involving the LL18 kernel on an SGI Challenge machine. We use a single Challenge processor, which incorporates a 16-Kbyte first-level cache, and a 1-Mbyte second-level cache with 128-byte cache lines. Both levels of the cache are direct-mapped. We perform one-dimensional partitioning to eliminate conflicts in the second-level cache because the miss latency to memory is much greater than the miss latency between the first and second levels of the cache. However, the tiled code is modified to exploit reuse in the first-level cache, i.e., a small tile size is used. Figure 8 shows the execution times measured for the original LL18 loop nest sequence, the fused version, and the tiled version, comparing the effect of various amounts of padding with a direct application of cache partitioning. As before, only cache partitioning is able to guarantee the best performance, even for the original loop nests prior to any locality enhancement.

Our final set of results demonstrate the effectiveness of cache partitioning in parallel execution. Figure 9 shows the speedup obtained on a KSR2 multiprocessor for the fused LL18 loop nests with data size 512×512. The results based on cache partitioning are compared with results obtained for some values of padding. As before, cache partitioning consistently provides the best performance. Furthermore, as more processors are used, the relative impact of misses due to conflicts grows, and the importance of cache partitioning increases.

In summary, our experimental results demonstrate the effectiveness of cache partitioning for locality-enhancing transformations for both sequential and parallel execution. Given the current trend of increasing processor speeds relative to memory speeds, the potential improvements from enhancing locality with transformations such as fusion and tiling will be greater for future processors. Cache partitioning ensures that the full benefit is actually obtained.

5 Related Work

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

Temam et al. [14] study conflicts arising from array references in loop nests typical of scientific applications. They analyze specific instances of self- and cross-conflicts, and suggest the use of padding and careful placement of arrays in memory. However, no detailed methodology is described for resolving conflicts.

Bacon et al. [2] discuss a method to determine the amount of padding needed to avoid cache and TLB mapping conflicts among individual array references in the innermost loop of a loop nest. Their method is heuristic, involving a search for an appropriate value of padding for each array. The method has been applied to one example resulting in a slight improvement in performance. Furthermore, their approach is not appropriate for locality-enhancing loop transformations, as it does not consider data reuse in outer loops, and therefore cannot prevent cache conflicts for reusable data.

Memory alignment [1] is another data-reorganization technique which aligns data in memory to cache line boundaries in a effort to contain individual data items within a single cache line whenever possible. While this can reduce cache traffic and decrease the potential for conflicts for small data sets, the impact is not significant for large data sets, such as arrays in scientific loop nests, because the cache line size is relatively small.

Lebeck and Wood [9] present a case study of improving performance through a variety of techniques including data transformations such as padding and memory alignment.

However, these transformations are discussed in the context of programmer tuning of application performance. There is no discussion of how such transformations may be incorporated into a compiler.

6 Concluding Remarks

This paper has described cache partitioning, a technique for eliminating cache conflicts among arrays referenced in parallel loop nests. Eliminating conflicts is particularly important to maximize the benefit of locality-enhancing loop transformations, which require that reusable data remain in the cache. Failure to resolve conflicts can render such transformations ineffective.

We presented experimental results from a KSR multiprocessor and an SGI Challenge machine. The results demonstrated the effectiveness of cache partitioning in eliminating cache conflicts and in realizing the full potential of two locality improvement techniques, loop fusion and tiling.

Cache partitioning is a significant improvement over previous approaches such as padding because it consistently ensures that conflicts are eliminated, including those conflicts which diminish the benefit of exploiting data reuse carried by outer loops. Furthermore, it simplifies the choice of locality optimization parameters, such as tile size. Cache partitioning is a suitable technique for inclusion in a compiler to ensure that the full benefit from locality optimization is realized. It requires no hardware support and incurs low memory space overhead in most applications.

The technique of cache partitioning can be extended to deal with multiple levels of cache by incorporating additional constraints on array starting addresses to prevent conflicts in all cache levels.

References

[1] 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.

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

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

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

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

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

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

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

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

[10] D. Lenoski et al. The DASH prototype: Implementation and performance. In *Proc. 19th Intl. Symp. on Computer Architecture*, pages 92–103, Gold Coast, Australia, May 1992.

[11] N. Manjikian and T. Abdelrahman. Fusion of loops for parallelism and locality. In *Proc. 1995 Intl. Conf. on Parallel Processing*, August 1995. To appear.

[12] 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.

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

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

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

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

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

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