# Inter-array Data Regrouping

Chen Ding Ken Kennedy

Rice University, Houston TX 77005, USA

**Abstract.** As the speed gap between CPU and memory widens, memory hierarchy has become the performance bottleneck for most applications because of both the high latency and low bandwidth of direct memory access. With the recent introduction of latency hiding strategies on modern machines, limited memory bandwidth has become the primary performance constraint and, consequently, the effective use of available memory bandwidth has become critical. Since memory data are transferred one cache block at a time, improving the utilization of cache blocks can directly improve memory bandwidth utilization and program performance. However, existing optimizations do not maximize cache-block utilization because they are *intra-array*; that is, they improve only data reuse within single arrays, and they do not group useful data of multiple arrays into the same cache block. In this paper, we present *inter-array data regrouping*, a global data transformation that first splits and then selectively regroups all data arrays in a program. The new transformation is optimal in the sense that it exploits inter-array cache-block reuse when and only when it is always profitable. When evaluated on real-world programs with both regular contiguous data access, and irregular and dynamic data access, inter-array data regrouping transforms as many as 26 arrays in a program and improves the overall performance by as much as 32%.

## 1 Introduction

As modern single-chip processors have increased the rate at which they execute instructions, performance of the memory hierarchy has become the bottleneck for most applications. In the past, the principal challenge in memory hierarchy management was in overcoming latency, but computation blocking and data prefetching have ameliorated that problem significantly. As exposed memory latency is reduced, the bandwidth constraint becomes dominant because the limited memory bandwidth restricts the rate of data transfer between memory and CPU regardless of the speed of processors or the latency of memory access. Indeed, we found in an earlier study that the bandwidth needed to achieve peak performance levels on intensive scientific applications is up to 10 times greater than that provided by the memory system[7]. As a result, program performance is now limited by its effective bandwidth; that is, the rate at which operands of a computation are transferred between CPU and memory.

The primary software strategy for alleviating the memory bandwidth bottleneck is cache reuse, that is, reusing the buffered data in cache instead of accessing memory directly. Since cache consists of non-unit cache blocks, sufficient use of cache blocks becomes critically important because low cache-block utilization leads directly to both low memory-bandwidth utilization and low cache utilization. For example for cache blocks of 16 numbers, if only one number is useful in each cache block, 15/16 or 94% of memory bandwidth is wasted, and furthermore, 94% of cache space is occupied by useless data and only 6% of cache is available for data reuse.

A compiler can improve cache-block utilization, or equivalently, cache-block spatial reuse, by packing useful data into cache blocks so that all data elements in a cache block are consumed before it is evicted. Since a program employs many data arrays, the useful data in each cache block may come from two sources: the data within one array, or the data from multiple arrays. Cache-block reuse within a single array is often inadequate to fully utilize cache blocks. Indeed, in most programs, a single array may never achieve full spatial reuse because data access cannot always be made contiguous in every part of the program. Common examples are programs with regular, but high dimensional data, and programs with irregular and dynamic data. When non-contiguous access to a single array is inevitable, the inclusion of useful data from other arrays can directly increase cache-block reuse.

This paper presents *inter-array data regrouping*, a global data transformation that first splits and then selectively regroups all data arrays in a program. Figure 1 gives an example of this transformation. The left-hand side of the figure shows the example program, which traverses a matrix first by rows and then by columns. One of the loops must access non-contiguous data and cause low cache-block utilization because only one number in each cache block is useful. Inter-array data regrouping combines the two arrays as shown in the right-hand side of Figure 1. Assuming the first data dimension is contiguous in memory, the regrouped version guarantees at least two useful numbers in each cache block regardless the order of traversal.

```
Array a[N,N], b[N,N]

// row-by-row traversal
For j=1, N
  For i=1, N
    F( a[i,j], b[i,j] )
  End for
End for

// column-by-column traversal
For i=1, N
  For j=1, N
    G( a[i,j], b[i,j] )
  End for
End for
```

```
Array c[2,N,N]

// row-by-row traversal
For j=1, N
  For i=1, N
    F( c[1,i,j], c[2,i,j] )
  End for
End for

// column-by-column traversal
For i=1, N
  For j=1, N
    G( c[1,i,j], c[2,i,j] )
  End for
End for
```

**Fig. 1.** Example of Inter-array Data Regrouping

In addition to improving cache spatial reuse, data regrouping also reduces the page-table (TLB) working set of a program because it merges multiple arrays into a single one. Otherwise, a program may touch too many arrays, causing overflow of TLB. On modern machines, TLB misses are very harmful to performance because CPU cannot continue program execution during a TLB miss.

Inter-array data regrouping can also improve communication performance of shared-memory parallel machines. On these machines, cache blocks are the basis of data consistency and consequently the unit of communication among parallel processors. Good cache-block utilization enabled by inter-array data regrouping can amortize the latency of communication and fully utilize communication bandwidth.

The rest of the paper is organized as follows. The next section formulates the problem of inter-array data regrouping and presents its solution. Section 3 evaluates the transformation on two well-known kernels and two real-world applications. Section 4 discusses related work, and Section 5 concludes.

## 2 Inter-array Data Regrouping

This section first describes the necessary program analysis for data regrouping, then gives the regrouping algorithm, proves its optimality, and finally discusses three extensions of the regrouping problem and presents their solutions as well.

### 2.1 Program Analysis

Given a program, a compiler identifies in two steps all opportunities of inter-array data regrouping. The first step partitions the program into a sequence of computation phases. A computation phase is defined as a segment of the program that accesses data larger than cache. A compiler can automatically estimate the amount of data access in loop structures with the technique given by Ferrante et al[8].

The second step of the analysis identifies sets of compatible arrays. Two arrays are compatible if their sizes differ by at most a constant, and if they are always accessed in the same order in each computation phase. For example, the size of array $A(3, N)$ is compatible with $B(N)$ and with $B(N-3)$ but not with $C(N/2)$ or $D(N, N)$. The access order from $A(1)$ to $A(N)$ is compatible with $B(1)$ to $B(N)$ but not with the order from $C(N)$ to $C(1)$ or from $D(1)$ to $D(N/2)$. The second criterion does allow compatible arrays to be accessed differently in different computation phases, as long as they have the same traversal order in the same phase[1].

[1] In general, the traversal orders of two arrays need not to be the same as long as they maintain a consistent relationship. For example, array $A$ and $B$ have consistent traversal order if whenever $A[i]$ is accessed, $B[f(i)]$ is accessed, where $f(x)$ is a one-to-one function.

The second step requires identifying the data access order within each array. Regular programs can be analyzed with various forms of array section analysis. For irregular or dynamic programs, a compiler can use the data-indirection analysis described by Ding and Kennedy (in Section 3.2 of [6]).

The other important job of the second step is the separation of arrays into the smallest possible units, which is done by splitting constant-size data dimensions into multiple arrays. For example, $A(2, N)$ is converted into $A1(N)$ and $A2(N)$.

After the partitioning of computation phases and compatible arrays, the formulation of data regrouping becomes clear. First, data regrouping transforms each set of compatible arrays separately because grouping incompatible arrays is either impossible or too costly. Second, a program is now modeled as a sequence of computation phases each of which accesses a subset of compatible arrays. The goal of data regrouping is to divide the set of compatible arrays into a set of new arrays such that the overall cache-block reuse is maximized in all computation phases.

### 2.2 Regrouping Algorithm

We now illustrate the problem and the solution of data regrouping through an example of a hydrodynamics simulation program, which computes the movement of particles in some three-dimensional space. Table 1 lists the six major computation phases of the program as well as the attributes of particles used in each phase. Since the program stores an attribute of all particles in a separate array, different attributes do not share the same cache block. Therefore, if a computation phase uses $k$ attributes, it needs to load in $k$ cache blocks when it accesses a particle.

**Table 1.** Computation phases of a hydrodynamics simulation program

| | Computation phases | Attributes accessed |
|---|---|---|
| 1 | *constructing interaction list* | `position` |
| 2 | *smoothing attributes* | `position, velocity, heat, derivate, viscosity` |
| 3 | *hydrodynamic interactions 1* | `density, momentum` |
| 4 | *hydrodynamic interactions 2* | `momentum, volume, energy, cumulative totals` |
| 5 | *stress interaction 1* | `volume, energy, strength, cumulative totals` |
| 6 | *stress interaction 2* | `density, strength` |

Combining multiple arrays can reduce the number of cache blocks accessed and consequently improve cache-block reuse. For example, we can group *position* and *velocity* into a new array such that the $i$th element of the new array contains the position and velocity of the $i$th particle. After array grouping, each particle reference of the second phase accesses one fewer cache blocks since *position* and *velocity* are now loaded by a single cache block. In fact, we can regroup all five arrays used in the second phase and consequently merge all attributes into a single cache block (assuming a cache block can hold five attributes).

However, excessive grouping in one phase may hurt cache-block reuse in other phases. For example, grouping *position* with *velocity* wastes a half of each cache block in the first phase because the *velocity* attribute is never referenced in that phase.

The example program shows two requirements for data regrouping. The first is to fuse as many arrays as possible in order to minimize the number of loaded cache blocks, but at the same time, the other requirement is not to introduce any useless data through regrouping. In fact, the second requirement mandates that two arrays should not be grouped unless they are always accessed together. Therefore, the goal of data regrouping is to partition data arrays such that (1) two arrays are in the same partition only if they are always accessed together, and (2) the size of each partition is the largest possible. The first property ensures no waste of cache, and the second property guarantees the maximal cache-block reuse.

Although condition (1) might seem a bit restrictive in practice, we note that many applications use multiple fields of a data structure array together. Our algorithm will automatically do what the Fortran 77 hand programmer does to simulate arrays of data structures: implement each field as a separate array. Thus it should be quite common for two or more arrays to always be accessed together. In Section 2.4 we discuss methods for relaxing condition (1) at the cost of making the analysis more complex.

The problem of optimal regrouping is equivalent to a set-partitioning problem. A program can be modeled as a set and a sequence of subsets where the set represents all arrays and each subset models the data access of a computation phase in the program.

Given a set and a sequence of subsets, we say two elements are *buddies* if for any subset containing one element, it must contain the other one. The *buddy* relation is reflexive, symmetric, and transitive; therefore it is a partition. A buddy partitioning satisfies the two requirements of data regrouping because (1) all elements in each partition are buddies, and (2) all buddies belong to the same partition. Thus the data regrouping problem is the same as finding a partitioning of buddies.

The buddy partitioning can be solved with efficient algorithms. For example, the following partitioning method uses set memberships for each array, that is, a bit vector whose entry $i$ is 1 if the array is accessed by the $i$th phase. The method uses a radix sort to find arrays with the same set memberships, i.e. arrays that are always accessed together. Assuming a total of $N$ arrays and $S$ computation phases, the time complexity of the method is $O(N * S)$. If a bit-vector is used for $S$ in the actual implementation, the algorithm runs in $O(N)$ vector steps. In this sense, the cost of regrouping is linear to the number of arrays.

### 2.3 Optimality

Qualitatively, the algorithm groups two arrays when and only when it is always profitable to do so. To prove, consider on the one hand, data regrouping never includes any useless data into cache, so it is applied only when profitable; on the other hand, whenever two arrays can be merged without introducing useless data, they are regrouped by the algorithm. Therefore, *data regrouping exploits inter-array spatial reuse when and only when it is always profitable.*

Under reasonable assumptions, the optimality can also be defined quantitatively in terms of the amount of memory access and the size of TLB working set. The key link between an array layout and its overall data access is the concept called *iteration footprint*, which is the number of distinct arrays accessed by one iteration of a computation phase. Assuming an array element is smaller than a cache block but an array is larger than a virtual memory page, then the iteration footprint is equal to the number of cache blocks and the number of pages accessed by one iteration. The following lemma shows that data regrouping minimizes the iteration footprint.

**Lemma 1.** *Under the restriction of no useless data in cache blocks, data regrouping minimizes the iteration footprint of each computation phase.*

*Proof.* After buddy partitioning, two arrays are regrouped when and only when they are always accessed together. In other words, two arrays are combined when and only when doing so does not introduce any useless data. Therefore, for any computation phase after regrouping, no further array grouping is possible without introducing useless data. Thus, the iteration footprint is minimal after data regrouping.

The size of a footprint directly affects cache performance because the more distinct arrays are accessed, the more active cache blocks are needed in cache, and therefore, the more chances of premature eviction of useful data caused by either limited cache capacity or associativity. For convenience, we refer to both cache capacity misses and cache interference misses collectively as *cache overhead misses*. It is reasonable to assume that the number of cache overhead misses is a non-decreasing function on the number of distinct arrays. Intuitively, a smaller footprint should never cause more cache overhead misses because a reduced number of active cache blocks can always be arranged so that their conflicts with cache capacity and with each other do not increase. With this assumption, the following theorem proves that a minimal footprint leads to minimal cache overhead.

**Theorem 2.** *Given a program of $n$ computation phases, where the total number of cache overhead misses is a non-decreasing function on the size of its iteration footprint $k$, then data regrouping minimizes the total number of overhead misses in the whole program.*

*Proof.* : Assuming the number of overhead misses in the $n$ computation phases is $f_1(k_1), f_2(k_2), \ldots, f_n(k_n)$, then the total amount of memory re-transfer is proportional to $f_1(k_1) + f_2(k_2) + \ldots + f_n(k_n)$. According to the previous lemma, $k_1, k_2, \ldots, k_n$ are the smallest possible after regrouping. Since all functions are non-decreasing, the sum of all cache overhead misses is therefore minimal after data regrouping.

The assumption made by the theorem covers a broad range of data access patterns in real programs, including two extreme cases. The first is the worst extreme, where no cache reuse happens, for example, in random data access. The total number of cache misses is in linear proportion to the size of the iteration footprint since each data access causes a cache miss. The other extreme is the case of perfect cache reuse where no cache overhead miss occurs, for example, in contiguous data access. The total number of repeated memory transfer is zero. In both cases, the number of cache overhead misses is a non-decreasing function on the size of the iteration footprint. Therefore, data regrouping is optimal in both cases according to the theorem just proved.

In a similar way, data regrouping minimizes the overall TLB working set of a program. Assuming arrays do not share the same memory page, the size of the iteration footprint, i.e. the number of distinct arrays accessed by a computation phase, is in fact the size of its TLB working set. Since the size of TLB working set is a non-decreasing function over the iteration footprint, the same proof can show that data regrouping minimizes the overall TLB working set of the whole program.

A less obvious benefit of data regrouping is the elimination of useless data by grouping only those parts that are used by a computation phase of a program. The elimination of useless data by array regrouping is extremely important for applications written in languages with data abstraction features, as in, for example, C, C++, Java and Fortran 90. In these programs, a data object contains lots of information, but only a fraction of it is used in a given computation phase.

In summary, the regrouping algorithm is optimal because it minimizes all iteration footprints of a program. With the assumption that cache overhead is a non-decreasing function over the size of iteration footprints, data regrouping achieves maximal cache reuse and minimal TLB working set.

### 2.4 Extensions

The previous section makes two restrictions in deriving the optimal solution for data regrouping. The first is disallowing any useless data, and the second is assuming a static data layout without dynamic data remapping. This section relaxes these two restrictions and gives modified solutions. In addition, this section expands the scope of data regrouping to minimizing not only memory reads but also memory writebacks.

**Allowing Useless Data** The base algorithm disallows regrouping any two arrays that are not always accessed together. This restriction may not always be desirable, as in the example program in the left-hand side of Figure 2, where array $A$ and $B$ are accessed together for in the first loop, but only $A$ is accessed in the second loop. Since the first loop is executed 100 times as often as the second loop, it is very likely that the benefit of grouping $A$ and $B$ in the first exceeds the overhead of introducing redundant data in the second. If so, it is profitable to relax the prohibition on no useless data.

**Allowing useless data**

```
for step=1, t
   for i=1, N
     Foo(A[i], B[i])
   end for
end for

for i=1, N
   Bar(A[i])
end for
```

**Allowing dynamic data remapping**

```
for step=1, t
   for i=1, N
     Foo(A[i], B[i])
   end for
end for

for step=1, t
   for i=1, N
     Bar(A[i])
   end for
end for
```

**Fig. 2.** Examples of extending data regrouping

However, the tradeoff between grouping fewer arrays and introducing useless data depends on the precise measurement of the performance gain due to data regrouping and the performance loss due to redundant data. Both the benefit and cost are program and machine dependent and neither of them can be statically determined. One practical remedy is to consider only frequently executed computation phases such as the loops inside a time-step loop and to apply data regrouping only on them.

When the exact run-time benefit of regrouping and the overhead of useless data is known, the problem of optimal regrouping can be formulated with a weighted, undirected graph, called a *data-regrouping graph*. Each array is a node in the graph. The weight of each edge is the run-time benefit of regrouping the two end nodes minus its overhead. The goal of data regrouping is to pack arrays that are most beneficial into the same cache block. However, the packing problem on a data-regrouping graph is NP-hard because it can be reduced from the G-partitioning problem[12].

**Allowing Dynamic Data Regrouping** Until now, data regrouping uses a single data layout for the whole program. An alternative strategy is to allow dynamic regrouping of data between computation phases so that the data layout of a particular phase can be optimized without worrying about the side effects in other phases. For example, in the example program in the right-hand side of Figure 2, the best strategy is to group $A$ and $B$ at the beginning of the program and then separate these two arrays after the first time-step loop.

For dynamic data regrouping to be profitable, the overhead of run-time data movement must not exceed its benefit. However, the exact benefit of regrouping and the cost of remapping are program and machine dependent and consequently cannot be determined by a static compiler. As in the case of allowing useless data, a practical approach is to apply data regrouping within different time-step loops and insert dynamic data remapping in between.

When the precise benefit of regrouping and the cost of dynamic remapping is known, the problem can be formulated in the same way as the one given by Kremer[13]. In his formulation, a program is separated into computation phases. Each data layout results in a different execution time for each phase plus the cost of changing data layouts among phases. The optimal layout, either dynamic or static, is the one that minimizes the overall execution time. Without modification, Kremer's formulation can model the search space of all static or dynamic data-regrouping schemes. However, as Kremer has proved, finding the optimal layout is NP-hard. Since the search space is generally not large, he successfully used 0-1 integer programming to find the optimal data layout. The same method can be used to find the optimal data regrouping when dynamic regrouping is allowed.

**Minimizing Data Writebacks** On machines with insufficient memory bandwidth, data writebacks impede memory read performance because they consume part of the available memory bandwidth. To avoid unnecessary writebacks, data regrouping should not mix the modified data with the read-only data in the same cache block.

To totally avoid writing back read-only data, data regrouping needs an extra requirement. Two arrays should not be fused if one of them is read-only and the other is modified in a computation phase. The new requirement can be easily enforced by a simple extension. For each computation phase, split the accessed arrays into two disjoint subsets: the first is the set of read-only arrays and the second is the modified arrays. Treat each sub-set as a distinctive computation phase and then apply the partitioning. After data regrouping, two arrays are fused if and only if they are always accessed together, and the type of the access is either both read-only or both modified. With this extension, data regrouping finds the largest subsets of arrays that can be fused without introducing useless data or redundant writebacks.

When redundant writebacks are allowed, data regrouping can be more aggressive by first combining data solely based on data access and then separating read-only and read-write data within each partition. The separation step is not easy because different computation phases read and write a different set of arrays. The general problem can be modeled with a weighted, undirected graph, in which each array is a node and each edge has a weight labeling the combined effect of both regrouping and redundant writebacks. The goal of regrouping is to pack nodes into cache blocks to maximize the benefit. As in the case of allowing useless data, the packing problem here is also NP-hard because it can be reduced from the G-partitioning problem[12].

## 3 Evaluation

### 3.1 Experimental Design

We evaluate data regrouping both on regular programs with contiguous data traversal and on irregular programs with non-contiguous data access. In each class, we use a kernel and a full application. Table 2 and Table 3 give the description of these four applications, along with their input size. We measure the overall performance except *Moldyn*, for which we measure only the major computation subroutine *compute_force*. The results of *Moldyn* and *Magi* were partially reported in [6] where data regrouping was used as a preliminary transformation to data packing.

**Table 2.** Program description

| name | description | access pattern | source | No. lines |
|---|---|---|---|---|
| *Moldyn* | molecular dynamics simulation | irregular | CHAOS group | 660 |
| *Magi* | particle hydrodynamics | irregular | DoD | 9339 |
| *ADI* | alternate-direction integration | regular | standard kernel | 59 |
| *Sweep3D* | nuclear simulation | regular | DOE | 2105 |

**Table 3.** Data inputs

| application | input size | source of input | exe. time |
|---|---|---|---|
| *Moldyn* | 4 arrays, 256K particles, 1 iteration | random initialization | 53.2 sec |
| *Magi* | 84 arrays, 28K particles, 253 cycles | provided by DoD | 885 sec |
| *ADI* | 3 arrays, 1000x1000 per array , 10 iterations | random initialization | 2.05 sec. |
| *Sweep3D* | 29 arrays, 50x50x50 per array | provided by DoE | 56 sec. |

The test programs are measured on a single MIPS R10K processor of SGI Origin2000, which provides hardware counters that measure cache misses and other hardware events with high accuracy. The processor has two caches: the first-level (L1) cache is 32KB with 32-byte cache blocks and the second-level (L2) cache is 4MB with 128-byte cache blocks. Both are two-way set associative. The processor achieves good latency hiding through dynamic, out-of-order instruction issue and compiler-directed prefetching. All applications are compiled with the highest optimization flag except *Sweep3D* and *Magi*, where the user specified -O2 to preserve numerical accuracy. Prefetching is turned on in all programs.

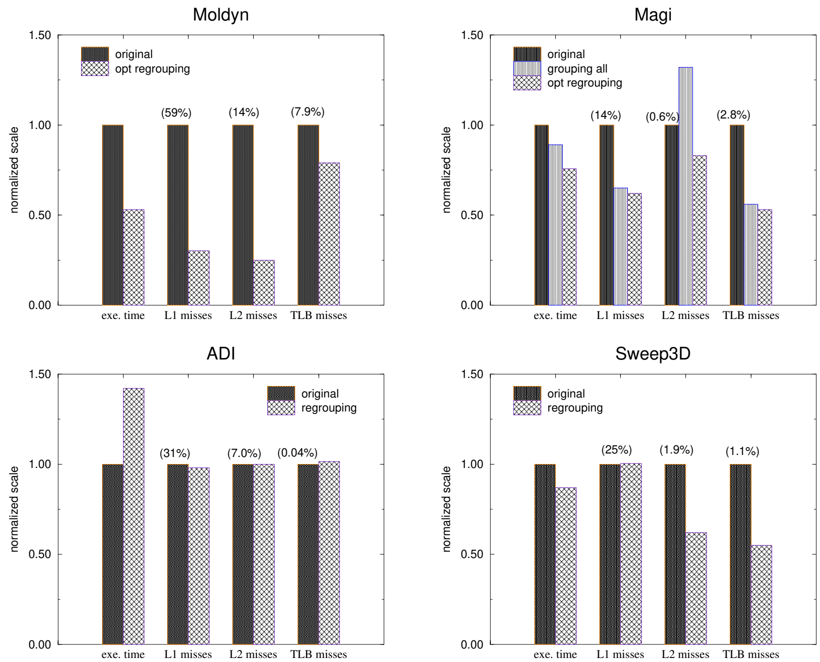


**Fig. 3.** Effect of Data Regrouping

### 3.2 Effect of Data Regrouping

Each of the four graphs of Figure 3 shows the effect of data regrouping on one of the applications; it shows the changes in four pairs of bars including the execution time, the number of L1 and L2 cache misses, and the number of TLB misses. The first bar of each pair is the normalized performance (normalized to 1) of the original program, and the second is the result after data regrouping. Application *Magi* has one more bar in each cluster: the first and the third are the performance before and after data regrouping, the middle bar shows the performance of grouping all data arrays. The graphs also report the miss rate of the base program, but the reduction is on the number of misses, not on the miss rate. The miss rate here is computed as the number of misses divided by the number of memory loads.

Data regrouping achieves significant speedups on the two irregular applications, *Moldyn* and *Magi*, as shown by the upper two graphs in Figure 3. Since both applications mainly perform non-contiguous data access, data regrouping improves cache-block reuse by combining data from multiple arrays. For *Moldyn* and *Magi* respectively, data regrouping reduces the number of L1 misses by 70% and 38%, L2 misses by 75% and 17%, and TLB misses by 21% and 47%. As a result, data regrouping achieves a speedup of 1.89 on *Moldyn* and 1.32 on *Magi*.

Application *Magi* has multiple computation phases, and not all arrays are accessed in all phases. Data regrouping fuses 26 compatible arrays into 6 arrays.

Blindly grouping all 26 arrays does not perform as well, as shown by the second bar in the upper-right graph: it improves performance by a much smaller factor of 1.12 and reduces L1 misses by 35% and TLB misses by 44%, and as a side effect, grouping-all increases L2 misses by 32%. Clearly, blindly grouping all data is sub-optimal.

The lower two graphs in Figure 3 show the effect of data regrouping on the two regular programs. The first is the *ADI* kernel. *ADI* performs contiguous data access to three data arrays, and its page-table working set fits in TLB. Data regrouping fuses two arrays that are modified and leaves the third, read-only array unchanged. Since the program already enjoys full cache-block reuse and incurs no capacity misses in TLB, we did not expect an improvement from data regrouping. However, we did not expect a performance degradation. To our surprise, we found the program ran 42% slower after regrouping. The problem was due to the inadequate dependence analyzer of the SGI compiler in handling interleaved arrays after regrouping. We adjusted level of data grouping so that instead of changing $X(j,i)$ and $B(j,i)$ to $T(1,j,i)$ and $T(2,j,i)$, we let $X(j,i)$ and $B(j,i)$ to be $T(j,1,i)$ and $T(j,2,i)$. The new version of regrouping caused no performance slowdown and it in fact ran marginally faster than the base program.

*Sweep3D* is a regular application where the computation repeatedly sweeps through a three-dimensional object from six different angles, three of which are diagonal. A large portion of data access is non-contiguous. In addition, *Sweep3D* is a real application with a large number of scalars and arrays and consequently, its performance is seriously hindered by an excessive number of TLB misses (??% of miss rate). The data layout seem to be highly hand-optimized because any relocation of scalar and array declarations we tried ended up in an increase in TLB misses.

Currently, we perform a limited regrouping, which combines eight arrays into two new ones. The limited regrouping reduces the overall L2 misses by 38% and TLB misses by 45%. The execution time is reduced by 13%. The significant speedup achieved by data grouping demonstrates its effectiveness for highly optimized regular programs.

In summary, our evaluation has shown that data regrouping is very effective in improving cache spatial reuse and in reducing cache conflicts and TLB misses for both regular and irregular programs, especially those with a large number of data arrays and complex data access patterns. For the two real-world applications, data regrouping improves overall performance by factors of 1.32 and 1.15.

## 4 Related Work

Many data and computation transformations have been used to improve cache spatial reuse, but inter-array data regrouping is the first to do so by selectively combining multiple arrays. Previously existing techniques either do not combine multiple arrays or do so in an indiscriminate manner that often reduces cache-block reuse instead of increasing it.

Kremer[13] developed a framework for finding the optimal static or dynamic data layout for a program of multiple computation phases[13]. He did not consider array regrouping explicitly, but his formulation is general enough to include any such transformations, however, at the expense of being an NP-hard problem. Data regrouping simplifies the problem by allowing only those transformations that always improve performance. This simplification is desirable for a static compiler, which cannot quantify the negative impact of a data transformation. In particular, data regrouping takes a conservative approach that disallows any possible side effects and at the same time maximizes the potential benefit.

Several effective techniques have been developed for improving spatial reuse within single data arrays. For regular applications, a compiler can rearrange either the computation or data to employ stride-one access to memory. Computation reordering such as loop permutations was used by Gannon et al.[9], Wolf and Lam[18], and McKinley et al[16]. Without changing the order of computation, Mace developed global optimization of data "shapes" on an operation dag[15]. Leung studied a general class of data transformations—unimodular transformations— for improving cache performance[14]. Beckman and Kelly studied run-time data layout selection in a parallel library[2]. The combination of both computation and data restructuring was explored by Cierniak and Li[5] and then by Kandemir et al[11]. For irregular and dynamic programs, run-time data packing was used to improve spatial reuse by Ding and Kennedy[6]. None of these techniques addressed the selective grouping of multiple data arrays, neither did they exploit inter-array cache reuse. However, these techniques are orthogonal to data regrouping. In fact, they should be always preceded by data regrouping to first maximize inter-array spatial reuse, as demonstrated by Ding and Kennedy in [6] where they combined data regrouping with data packing.

Data placement transformations have long been used to reduce data interference in cache. Thabit packed simultaneously used data into non-conflicting cache blocks[17]. To reduce cache interference among array segments, a compiler can make them either well separated by padding or fully contiguous by copying. Data regrouping is different because it reorganizes not scalars or array segments but individual array elements. By regrouping elements into the same cache block, data grouping guarantees no cache interference among them. Another important difference is that data regrouping is conservative, and it does not incur any memory overhead like array copying and padding.

Data transformations have also been used to avoid false data sharing on parallel machines with shared memory. The transformations group together data that is local to each processor. Examples include those of Anderson et al.[1] and of Eggers and Jeremiassen[10]. Anderson et al. transformed a single loop nest at a time and did not fuse multiple arrays; Eggers and Jeremiassen transformed a single parallel thread at a time and fused all data it accesses. However, blindly grouping local data wastes bandwidth because not all local data is used in a given computation phase. Therefore, both data transformations are sub-optimal compared to data regrouping, which selectively fuses multiple arrays for maximal cache spatial reuse and cache utilization.

Besides the work on arrays, data placement optimizations have been studied for pointer-based data structures[3, 4]. The common approach is to first find data objects that are frequently accessed through profiling and then place them close to each other in memory. In contrast to data regrouping, they did not distinguish between different computation phases. As a result, these transformations are equivalent to grouping all frequently accessed objects in the whole program. As in the case of greedy regrouping by Eggers and Jeremiassen[10], the result is sub-optimal.

## 5 Conclusions

In this work, we have developed inter-array data regrouping, a global data transformation that maximizes overall inter-array spatial reuse in both regular and dynamic applications. The regrouping algorithm is compile-time optimal because it regroups arrays when and only when it is always profitable. When evaluated on both regular and irregular applications including programs involving a large number of arrays and multiple computation phases, data regrouping combines 8 to 26 arrays and improves overall performance by factors of 1.15 and 1.32. Similar regrouping optimization can also improve cache-block utilization on shared-memory parallel machines and consequently improve their communication performance.

The significant benefit of this work is that the automatic and effective data transformations enable a user to write machine-independent programs because a compiler can derive optimal data layout regardless the initial data structure specified by the user. Since the choice of global regrouping depends on the computation structure of a program, data regrouping is a perfect job for an automatic compiler, and as shown in this work, a compiler can do it perfectly.


### Acknowledgement

We'd like to thank anonymous referees of LCPC'99 and especially, Larry Carter, who corrected a technical error of the paper and helped in improving the discussion of the optimality of data regrouping.


## References


1. J. Anderson, S. Amarasinghe, and M. Lam. Data and computation transformation for multiprocessors. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, CA, July 1995.
2. O. Beckmann and P.H.J. Kelly. Efficient interprocedural data placement optimisation in a parallel library. In *Proceedings of the Fourth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, May 1998.

3. B. Calder, K. Chandra, S. John, and T. Austin. Cache-conscious data placement. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, San Jose, Oct 1998.
4. T.M. Chilimbi, B. Davidson, and J.R. Larus. Cache-conscious structure definition. In *Proceedings of SIGPLAN Conference on Programming Language Design and Implementation*, 1999.
5. M. Cierniak and W. Li. Unifying data and control transformations for distributed shared-memory machines. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, La Jolla, 1995.
6. C. Ding and K. Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation*, Atlanta, GA, May 1999.
7. C. Ding and K. Kennedy. Memory bandwidth bottleneck and its amelioration by a compiler. Technical report, Rice University, May 1999. Submitted for publication.
8. J. Ferrante, V. Sarkar, and W. Thrash. On estimating and enhancing cache effectiveness. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing, Fourth International Workshop*, Santa Clara, CA, August 1991. Springer-Verlag.
9. D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformations. In *Proceedings of the First International Conference on Supercomputing*. Springer-Verlag, Athens, Greece, June 1987.
10. Tor E. Jeremiassen and Susan J. Eggers. Reducing false sharing on shared memory multiprocessors through compile time data transformations. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 179–188, Santa Barbara, CA, July 1995.
11. M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. A matrix-based approach to the global locality optimization problem. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, 1998.
12. D. G. Kirkpatrick and P. Hell. On the completeness of a generalized matching problem. In *The Tenth Annual ACM Symposium on Theory of Computing*, 1978.
13. U. Kremer. *Automatic Data Layout for Distributed Memory Machines*. PhD thesis, Dept. of Computer Science, Rice University, October 1995.
14. S. Leung. Array restructuring for cache locality. Technical Report UW-CSE-96-08-01, University of Washington, 1996. PhD Thesis.
15. M.E. Mace. *Memory storage patterns in parallel processing*. Kluwer Academic, Boston, 1987.
16. K. S. McKinley, S. Carr, and C.-W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.
17. K. O. Thabit. *Cache Management by the Compiler*. PhD thesis, Dept. of Computer Science, Rice University, 1981.
18. M. E. Wolf and M. Lam. A data locality optimizing algorithm. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, Toronto, Canada, June 1991.

This article was processed using the LaTeX macro package with LLNCS style