Improving Effective Bandwidth through Compiler Enhancement of Global Cache Reuse

Chen Ding
Department of Computer Science
University of Rochester
Rochester, NY

Ken Kennedy
Department of Computer Science
Rice University
Houston, TX

Abstract

Reusing data in cache is critical to achieving high performance on modern machines because it reduces the impact of the latency and bandwidth limitations of direct memory access. To date, most studies of software memory hierarchy management have focused on the latency problem. However, today's machines are increasingly limited by insufficient memory bandwidth—on these machines, latency-oriented techniques are inadequate because they do not seek to minimize the total memory traffic over the whole program. This paper explores the potential for addressing bandwidth limitations by increasing global cache reuse—that is, reusing data across whole program and over the entire data collection. To this end, the paper explores a two-step global strategy. The first step fuses computations on the same data to enable the caching of repeated accesses. The second step groups data used by the same computation to bring about contiguous access to memory. While the first step reduces the frequency of memory accesses, the second step improves their efficiency. The paper demonstrates the effectiveness of this strategy and shows how to automate it in a production compiler.

1 Introduction

Although the problem of high memory latency in modern processors has been widely recognized, less attention has been paid to the effect of limited memory bandwidth. Over the past twenty years, CPU speed has improved by a factor of 6400, while memory bandwidth has increased by a factor of only 150[1]. Our earlier performance study found that typical scientific applications demand 3 to 10 times as much memory bandwidth as provided by a typical machine, and consequently, these applications can achieve at most 10% to 33% of peak CPU performance on average [11]. This indicates that substantive performance gains can be made on most applications if the memory bandwidth bottleneck can be alleviated[2].

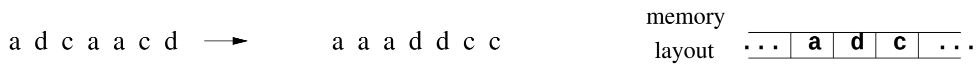
The principal method used to reduce the volume of memory traffic and thus alleviate the bandwidth bottleneck is data caching. Interestingly, although cache has been widely used for many years, its full potential has not yet been realized. The example in Figure 1 illustrates the opportunity for additional cache reuse. Part (a) of Figure 1 shows a sequence of 7 accesses to 3 data elements. Assuming a single-element cache with the LRU replacement, only the last access to a will find a buffered copy in cache, in which case, we credit the computation with a single cache reuse. On a perfect machine that can foresee all future computations, we could use the optimal replacement strategy, the Belady policy, which always keeps the next used data element in cache [5]. Belady would keep a in cache from the beginning and achieve two cache reuses. Still, the majority of memory access is not cached. Can this be improved upon?

In fact, the answer is "yes". The optimality of Belady policy is conditional: it assumes that the access sequence cannot be changed. If we can reorder data accesses, we can reuse data in a totally different manner. Part (b) of Figure 1 shows a reordered sequence where references to the same data are grouped together. The improved sequence has 4 cache reuses on a one-element cache with LRU replacement. This doubles the cache reuse achieved by Belady. In fact, the new sequence achieves the best caching behavior because each element is loaded only once.

In this paper, we seek improve data caching through program reorganization as described above. Our strategy consists of two steps. The first step, *computation fusion*, fuses computations on the same data so that when a piece of data is loaded in cache, we finish all its uses before evicting the data from cache. The clustering used in Figure 1(b) is an example of computation fusion. For caches with non-unit size cache blocks, computation fusion alone is not sufficient because it does not address the problem of data layout. To better utilize cache space, the second step, *data regrouping*, gathers data used by the same computation so that we populate cache exclusively with data used by that computa-

[1] We derived this estimate from historical data about CPU speed, memory pin count, and pin-bandwidth increases compiled by Burger et al[6].

[2] Common latency-hiding techniques such as prefetching cannot help with the bandwidth problem, because they do not reduce memory trafic. For example, when running a program with 10GB memory transfer on a machine with 1GB/s memory bandwidth, the execution would take at least 10 seconds even with infinite CPU speed and perfect prefetching. This time bound cannot be further reduced without first solving the bandwidth problem.



(a) example sequence (b) fusing accesses on the same data (c) grouping data used together

Figure 1. Example use of the two-step strategy

tion. Together, these two steps improve both the temporal and spatial locality of a program. Figure 1(c) shows the memory layout generated by data regrouping. The new access sequence and its data layout result in the best possible cache performance: not only are data elements loaded once and only once, but also they are loaded from contiguous memory locations.

The two-step strategy must be globally applied to enhance data reuse across the whole program. Computation fusion must recombine all procedures in a program and data grouping must re-shuffle the entire data layout. Since the two transformations are tightly coupled, they should not be manually applied because this would compromise the independence of procedures from their data layout. Existing automatic techniques, however, are not adequate. For example, loop blocking—a standard technique for increasing reuse—is usually applied to a single loop nest and thus cannot exploit data reuse among disjoint loops. Loop fusion combines multiple loops, but existing fusion methods are limited in their ability to fuse loops of different shape, such as non-perfectly nested loops and nests with different levels of nesting. Furthermore, previous fusion methods do not adequately address the problem of spatial locality, which becomes particularly difficult when a large amount of computation is fused together.

In the rest of this paper, we present two new program transformations that carry out the two-step strategy over the whole program and all its data. The first is *reuse-based loop fusion*, which fuses loops of different shapes. The second is *multi-level data regrouping*, which reorganizes arrays at various granularity. The next two sections describe these methods and explore their properties. Section 4 presents the implementation and evaluation of the combined strategy. Finally, Section 5 discusses related work and Section 6 concludes the presentation.

2 Global computation fusion

In this paper, we pursue computation fusion using a heuristic we call *sequential greedy fusion*—for every data reference from the beginning to the end of a program, we move it forward as far as possible toward the previous reference to the same data element. In a sense, this is the inverse of the Belady strategy. While Belady evicts data that have the furthest reuse, sequential greedy fusion executes the instruction that has the nearest reuse. In the next two sections, we first measure the potential of this heuristic through ideal simulation and describe a new source-level fusion transformation produces a reference pattern closer to the ideal.

2.1 Reuse-driven execution

Reuse-driven execution simulates sequential greedy fusion by reordering run-time instructions. We use simulation because it offers the maximal freedom in program reordering—it knows precise dependences and it permits transformations that are not restricted by source-level structures.

Given a program, we first instrument it to collect the execution trace. Only source-level instructions and data references are collected. Then we re-order instructions based on their issuing cycle on an ideal parallel machine with an unbounded number of processors. Finally we carry out reuse-driven execution following the algorithm in Figure 2. At each step, reuse-driven execution gives priority to later instructions that reuse the data of the current instruction. It employs a FIFO queue to sequentialize multiple reuse candidates.

```
function Main
    for each instruction i in the ideal
        parallel execution order
      enqueue i to ReuseQueue
      while ReuseQueue is not empty
        dequeue i from ReuseQueue
          if (i has not been executed)
            ForceExecute(i)
      end while
    end for
end Main

function ForceExecute(instruction j)
    while there exists un-executed instruction
        i that produces operands for j
      ForceExecute(i)
    end while
    execute j
    for each data t used by j
      find next instruction m that uses t
      enqueue m into ReuseQueue
    end for
end ForceExecute
```

Figure 2. Reuse-driven execution algorithm

To assess changes in program locality, we use a measure called *reuse distance* [12]. In a sequential execution, the reuse distance of a memory reference is the number of the *distinct* data items that appear between the current reference and the closest previous reference to the same data element. For example, in Figure 1(a), the reuse distance of the second access to d is two because two distinct data elements are accessed between the two uses of d. With a fully associative LRU cache, a memory reference hits in cache if and only if its reuse distance is smaller than cache size. Although the relation is not as definitive for a set-associative cache, it is generally true that the longer the reuse distance, the less the chance the cache reuse. We therefore measure the locality of a program by the histogram of all its reuse distances. A

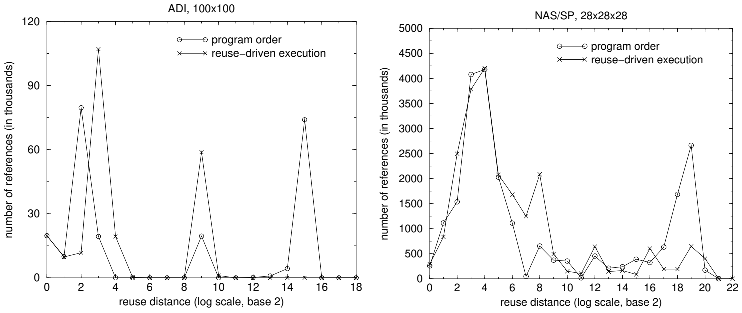


Figure 3. Effect of reuse-driven execution

detailed description of reuse distance and its measurement can be found in our technical report [12].

The effect of reuse-driven execution is shown in Figure 3 for a kernel program *ADI* and an application benchmark *NAS/SP* (Serial version 2.3); the former has 8 loops in 4 loop nests, and the latter has over 218 loops in 67 loop nests. In each figure, a point at (x, y) indicates that y thousands of memory references have a reuse distance between $[2^{(x-1)}, 2^x)$. Note that the horizontal axis is in log scale. The figure links discrete points into a curve to emphasize the elevated hills, where large portions of the memory references reside.

The two graphs in Figure 3 show that reuse-driven execution achieves a significant reduction in the number of long reuse distances. The reduction is approximately 30% for *ADI* and 60% for *SP*. Assuming that memory references with long reuse distances are the principal cause of cache misses, sequential greedy fusion would reduce the total number of misses by 30% to 60%. We also tested other programs—a kernel, *FFT*, and a full application, *DOE/Sweep3D*. Reuse-driven execution did not improve *FFT* (a small percentage of reuse distances is actually lengthened), but it reduced the number of long reuse distances by about 60% in *DOE/Sweep3D*. The complete results and a more accurate characterization can be found in our technical report [12].

In summary, reuse-driven execution demonstrates the effectiveness of sequential greedy fusion on programs with a large number of loops such as *NAS/SP* and *DOE/Sweep3D*. The next section shows how to approximate the effect of sequential greedy fusion at the source level. The evaluation section will compare the result of source-level fusion with that of reuse-driven execution.

2.2 Reuse-based loop fusion

In many applications, especially scientific programs, loops contain most of the data accesses and hence, most of the opportunities data reuse. Applying sequential greedy fusion means fusing each loop with the preceding loop with which it shares data. Although loop fusion has been widely studied, its use has been limited in real programs that contain loops of different shapes, such as single statements (loops with zero dimension), loops of a different number of dimensions, and non-perfectly nested loops, because it is not immediately obvious what it means to fuse loops of different shapes.

Reuse-based fusion uses a general heuristic based on data reuse: given two loops of arbitrary shape, we merge the iterations that use the same data. The focus on data access allows loop fusion to be based on the content of the loop rather than on its control structure. Given two loops, we categorize their data sharing into three cases and apply transformations correspondingly as follows.

- *Loop fusion and alignment, when data are shared between iterations of the candidate loops.* We interleave the iterations of the two loops and align them by either shifting the iterations of the second loop down to preserve data dependences, or, if data dependences permit, shifting these iterations up to bring together data reuse.
- *Loop embedding, when data are shared between one loop and an iteration of the other loop.* We embed the first loop as one iteration in the second loop. We schedule the embedded iteration with its data-sharing counterpart.
- *Iteration reordering, when candidate loops are not wholly fusible.* We break the two loops into individual iterations and fuse those iterations that are fusible. Examples of iteration reordering are loop splitting and loop reversal.

Figure 4 gives the basic algorithm for single-level fusion. For each statement p, the algorithm finds the closest predecessor q that shares data access with p. It then examines the data reuse to determine the proper use of loop fusion, loop alignment, loop embedding and iteration reordering. Once two loops are fused, the fusion process is re-applied to the fused loop because it now accesses a larger set of data and may require further fusion with its predecessors. Although not shown in Figure 4, the actual algorithm has provisions to control the cost of recursive fusion. The implementation of this algorithm will be discussed in more detail in Section 4.1, including the handling of branches and procedures.

```
SingleLevelFusion
    for each statement p in program order
      find latest preceding loop q that
         shares data with p
      if (p share data w/ an iteration of q)
         embed p into q by loop embedding
      else begin
         find the smallest alignment factor
           that satisfies data dependences
         apply iteration reordering if necessary
         if (legal alignment is found)
           fuse p into q by loop alignment
           recursively apply fusion on q
         end if
      end if
    end for
```

Figure 4. Single-level fusion algorithm

The example in Figure 5 illustrates reuse-based loop fusion. The code on the left-hand side of Figure 5 is a program with two loops that both access array A. They cannot be fused directly because two intervening statements also access parts of A. The fusion algorithm examines each statement in program order and embeds the two single statements into the first loop. After loop embedding, two loops are still not fusible because $A[1]$ is assigned by the last iteration of the first loop but used by the first iteration of the second loop. Therefore, iteration reordering is applied to peel off the first iteration of the second loop so that the remaining iterations can be fused with the first loop. Finally, loop alignment shifts up the iterations of the second loop so that they can directly reuse $A[i-1]$. The fused program is shown in the right-hand side of Figure 5.

```
for i=2, N
  A[i]=f(A[i-1])
end for

A[1]=A[N]
A[2]=0.0

for i=3, N
  B[i]=g(A[i-2])
end for
```

```
for i=2, N
  A[i]=f(A[i-1])
  if (i==3)
    A[2]=0.0
  else if (i==N)
    A[1]=A[N]
  end if
  if (i>2 and i<N)
    B[i+1]=g(A[i-1])
  end if
end for
B[3]=g(A[1])
```

Figure 5. Examples of reuse-based fusion

Reuse-based loop fusion adds significant instruction overhead because of the branch statements it inserts. For programs that already use cache well, this overhead may degrade performance, particularly on processors with limited branching capability. However, on programs whose performance is seriously limited by memory hierarchy, the overhead of branching is likely to be hidden because the CPU spends most of its time waiting for memory. In addition, future processor generations will be better equipped to deal with branches because of advanced features such as predicated execution. The evaluation section will measure the overall effect of reuse-based fusion on a real machine.

2.3 Properties of reuse-based fusion

Pair-wise fusion This property is important because avoids the exponential cost of the fusion legality test. The problem is illustrated by the example program in Figure 6. The two loops shown cannot be fused because all iterations of the second loop depend on all iterations of the first loop due to the intervening statement. If we view the statement as a loop with zero dimension, then this is an example where any two of the three loops are fusible, but the group is not. Therefore, the fusion legality test is not transitive. Finding all fusible sets requires testing all loop groups and incurs a cost that is exponential in the number of loops. Reuse-based fusion, however, fuses loops incrementally. At each step, it examines only the closest pair of data-sharing loops and thus avoids legality test for arbitrary sets of loops, although it may miss some opportunities for fusion.

Bounded reuse distance In a fused loop, the reuse distance of all loop-variant accesses does not increase as the input size grows, that is, the fused loop can be cached by constant-size cache regardless the volume of the input data. The upper bound on reuse distance is $O(N_{arrays} * N_{loops})$, where N_{loops} is the number of loops that are fused, and N_{arrays} is the number of arrays accessed in the fused loop. This upper bound is tight because a worst-case example can be constructed as follows: the body of the first loop is `B(i)=A(i+1)`, next are N loops with a body `B(i)=B(i+1)`, finally is a loop with the body `A(i)=B(i)`. Since the two accesses to $A(i)$ must be separated by N iterations, the reuse distance can be no less than N. Therefore, the fusion algorithm achieves the tightest asymptotic upper bound on the length of reuse distances in fused loops.

```
for i=2, N
   A[i] = f(A[i-1])
end for
A[1] = A[N]
for i=2, N
   A[i] = g(A[i-1])
end for
```

Figure 6. Loops that are not fusible

Fast algorithm Assuming loop splitting at boundary elements is the only form of iteration reordering, the fusion algorithm in Figure 4 runs in $O(N * N' * A)$, where N is the number of program statements before fusion, N' is the number of loops after fusion, and A is the number of data arrays in the program. We expect that the number of program statements is much larger than the number of fused loops and program arrays. Furthermore, the latter two quantities probably do not grow in proportion to the number of statements. Therefore, the time complexity of reuse-based fusion in practice should be nearly linear in the in the size of the program. Other types of iteration reordering may add higher time complexity to the algorithm. However, boundary splitting is sufficient for test programs used in Section 4.

2.4 Multi-level fusion

For programs with multi-dimensional loops and arrays, the one-level fusion algorithm can be applied level by level as long as we first determine the nesting order. Figure 7 gives the algorithm for multi-level fusion, which seeks to minimize the number of fused loops at outer levels.

While all data structures and loop levels are used to determine the correctness of fusion, only large data structures are considered in determining the profitability of fusion. In multi-level fusion, the term *data dimension* denotes a data dimension of a large array, and the term *loop level* refers to only loops that iterate over a data dimension of a large array.

For each loop level starting from the outermost, *MultiLevelFusion* determines loop fusion at a given level, L, in three steps. The first step examines all data dimensions that are iterated by loops of this or deeper levels. For each data dimension, the algorithm performs hypothetical fusion and measures the number of loops after fusion. Then the second step picks the data dimension that yields the smallest number of fused loops. The actual transformation of interchange and fusion at level L happens at this step. Finally, the third step recursively applies *MultiLevelFusion* at the next loop level. Note that not all level-L loops iterate the same data dimension. Since loop interchange at the first step may not always succeed, some level-L loops may access a different data dimension. These loops will also be fused if they iterate the same data dimension. So the dimension s in the third step is not always the dimension s' found in the second step.

```
MultiFusion(S: data dimensions, L: current level)
   // 1. find the best data dimension for level L
   for each dimension s, test hypothetically
     LoopInterchange(s, L)
     apply SingleLevelFusion
     count the number of fused loops
   end for
   chose dimensiont s' with the fewest fused loops
   // 2. fuse loops for level L on dimension s'
   LoopInterchange(s', L)
   apply SingleLevelFusion at level L
   // 3. continue fusion at level L+1
   for each loop nest at level L
     MultiFusion(S-{s},L+1), where s is the
       data dimension iterated by the level-
L loop
   end for
end MultiFusion

LoopInterchange(s: data dimension, L: loop level)
     for each loop nest
       if (level t (>=L) iterates data dim s)
          interchange level t to L if possible
     end for
end LoopInterchange
```

Figure 7. Multi-level fusion algorithm

3 Global data regrouping

Although loop fusion reorders computations, it does not change data layout. Indeed, a fused loop usually includes a large volume of data access, which often causes wasted space within cache blocks and excessive interference among cache blocks. In this section, we overcome this problem by a method called *data regrouping*, which clusters data used by the same loop into adjacent memory locations. We first introduce our earlier work on single-level data regrouping and then extend it to grouping data of multiple granularity.

3.1 Single-level regrouping

To improve spatial reuse among global arrays, we previously introduced *inter-array data regrouping* [10], which works as follows. It first partitions a program into a sequence of computation phases, each of which accesses a data set that is larger than cache. Then, it classifies all data arrays into compatible groups based on their size and data access order. Arrays with small, constant size dimensions are split into multiple arrays.

Data regrouping is applied for each compatible array group. *Two arrays are grouped if and only if they are always accessed together.* The profitability is guaranteed because no useless data is introduced into cache blocks. In fact, this regrouping scheme achieves the best machine-independent data layout [10]. A limitation, however, is that it regroups data only at a single level. Here we extend regrouping to data of multiple granularities.

```
for i
  for j
    g(A[j,i],B[j,i])
  end for
  for j
    t(C[j,i])
  end for
end for
(a) Original program
```

```
for i
  for j
   g(D[1,j,1,i],D[2,j,1,i])
  end for
  for j
    t(D[j,2,i])
  end for
end for
(b) Transformed program
(assuming column-major order)
```

Figure 8. Example of multi-level regrouping

3.2 Multi-level regrouping

In many programs, especially after aggressive fusion, loops are not perfectly nested. Since data of varied granularity are used together, single-level array regrouping is not adequate to fully exploit spatial reuse among related data. Multi-level regrouping overcomes this limitation by grouping not only array elements but also array segments. The additional grouping at larger granularity can further reduce cache interference as well as the page-table working set.

Figure 8(a) gives an example of multi-level data regrouping. The program has an outer loop enclosing two inner loops. Elements of array A and B are used together in the first inner loop, and the columns of all three arrays are used together in iterations of the outer loop. Single-level regrouping cannot exploit these two levels of spatial reuse at the same time. To overcome this limitation, we group data at different granularities. In part (b), multi-level regrouping uses a new array D, which combines the first two arrays by their elements and all three arrays by their columns. After multi-level regrouping, the data access is contiguous in each iteration of the two inner loops (accessing $D[*, j, 1, i]$ and $D[j, 2, i]$) and each iteration of the outer loop (accessing $D[*, i]$), even though the loops are not perfectly nested.

It should be noted that programming languages like Fortran do not allow arrays of non-uniform dimensions like those of array D. In addition, the new array indexing created by source-level regrouping may confuse the back-end compiler and negatively affect its register allocation. However, both problems disappear when regrouping is applied, as it should be, by a back-end compiler.

The algorithm for multi-level regrouping is shown in Figure 9. The first step of *MultiLevelRegrouping* collects data accesses at all array dimensions. Two criteria are used to identify the set of arrays accessed at a given dimension. The first is necessary for the algorithm to be correct. The second does not affect correctness, but ensures that the algorithm counts only those memory references that represent significant data access volumes, that is, accessing the whole array. For each data dimension, the algorithm finds the set of accessed arrays in each computation phase. The second step of the algorithm applies one-level regrouping for each data dimension. The correctness of multi-level regrouping is stated in the following theorem, which is proved in Ding's

```
MutiLevelRegrouping
    // 1. find the arrays accessed in loops
    for each loop i and each array ref a
      for each data dimension d of a, find
          the set of arrays s such that
        1. each array in s is accessed at all
           dimensions higher than or equal to d
           by loop i and its outer loops, and
        2. each array of s is accessed at all
           other dimensions by inner loops of i
    end for
    // 2. partition ar-
rays for each data dimension
    for each data dimension d
        let S be the sets found in Step 1 for d
        let A be the set of all arrays
        OneLevelRegrouping(A, S, d)
    end for
end MultiLevelRegrouping

OneLevelRegrouping(A: all arrays, S: sub-
sets of A, d: current dimension)
    let N be the size of A
    // 1. construct a bit vector for each subset
    for each subset s in S
        construct a bit vector b of length N
        for i from 1 to N
          if (array a is in s) b[a]=1
          else b[a]=0
    end for; end for
    // 2. partition arrays
    sort all bit vectors using radix sort
    group arrays with the same bit vec-
tor at dim d
end OneLevelRegrouping
```

Figure 9. Multi-level regrouping algorithm

dissertation [9]. The proof shows that the grouping decision at a lower level (e.g., grouping array a with b) does not contradict the decision at a higher level (e.g., separating a and b).

Theorem 3.1 *If the algorithm in Figure 9 merges two arrays at data dimension d, the algorithm must also group these two arrays at all dimensions higher than d.*

4 Evaluation

4.1 Implementation

We have implemented computation fusion and data regrouping in a version of the D Compiler System at Rice University. The compiler performs whole program compilation given all source files of an input program. It uses a powerful value-numbering package to handle symbolic variables and expressions inside each subroutine and parameter passing between subroutines. It has a standard set of loop and dependence analysis, data flow analysis and interprocedural analysis.

Our compiler models the computation and data accesses of the input program in the following way. We restrict the accuracy of the model to ensure the fast time bound (given in Section 2.3) so that global fusion is practical for large programs.

- A program is a list of loop and non-loop statements, so is the body of each loop. A structured branch is treated as one meta-statement. A function call is either inlined or assumed to access all data.
- Each subscript position of an array reference is in one of the two forms: $A[i+t]$ and $A[t]$, where A is the variable name, i is a loop index, and t is a loop-invariant constant, otherwise we assume the subscript ranges over the whole data dimension.

For data access that does not conform to our simplified model, we use conservative approximations to guarantee the correctness of the transformation. More accurate representations such as affine array subscripts can be used, although at the expense of a slower algorithm. Regardless of the representation, our fusion algorithm using loop alignment, embedding and splitting is still applicable.

For each loop, the compiler summarizes its data access by its data footprint. For each dimension of an array, a data footprint describes whether the loop accesses the whole dimension, a number of elements on the border, or a loop-variant section (a range enclosing the loop index variable). Data dependence is tested by the intersection of footprints. The range information is also used to calculate the minimal alignment factor between loops.

An input program is processed by four preliminary transformations before applying loop fusion. The first is procedure inlining, which brings all computation loops into a single procedure. The next includes array splitting and loop unrolling, which expand data dimensions of a small constant size and loops that iterate those dimensions. The third step is loop distribution. Finally, the last step propagates constants into loop statements. Our compiler performs loop unrolling and constant propagation automatically. Currently, array splitting requires a user to specify the names, and inlining is done by hand; however, both transformations can be automated with additional implementation effort.

Loop fusion is carried out by applying the fusion algorithm in Figure 4 level by level from outermost to innermost. The current implementation calculates data footprints, aligns loops and schedules non-loop statements. Iteration reordering is not yet implemented but the compiler identifies the places where it is needed. Only one program, *Swim*, required splitting, which was done by hand.

For multi-dimensional loops, loop fusion reorders loop levels to maximize the benefit of outer-level fusion, following the algorithm in Figure 7. In our experiment, however, loop ordering was largely unnecessary, as computations were mostly symmetric. One exception was *Tomcatv*, where loop ordering (loop interchange) was performed by hand.

Code generation is based on mappings from the old iteration space to the fused iteration space. Currently, the code is generated by the Omega library [20], which has been integrated into the D compiler system [1]. Omega worked

name	source	input size	lines/loops/arrays
Swim	SPEC95	513x513	429/8/15
Tomcatv	SPEC95	513x513	221/18/7
ADI	self-written	2Kx2K	108/8/3
SP	NAS/NPB Serial v2.3	class B, 3 iterations	1141/218/15
Sweep3D	DOE	150x150x150	2105/67/6

Table 1. Description of test programs

well for small programs, where the compilation time was under one minute for all kernels. For the full application *SP*, however, code generation took four minutes for one-level fusion but one hour and a half for three-level fusion. In contrast, the fusion analysis took about two minutes for one-level fusion and four minutes for full fusion. A direct code generation scheme, given by Allen and Kennedy [4], is linear in the number of loop levels, there is currently no implementation in the D System.

The analysis for data regrouping is trivial with data footprints. After fusion, data regrouping is applied level by level on fused loops using the algorithm in Figure 9, with two modifications. First, SGI's compiler does a poor job when arrays are interleaved at the innermost data dimension. So the compiler groups arrays up to the second innermost dimension. This restriction may result in grouping in the less desired dimension, as in *Tomcatv*. The other restriction is due to the limitation of Fortran language, which does not allow non-uniform array dimensions. When multi-level regrouping produced non-uniform arrays, manual changes were made to disable regrouping at outer data dimensions.

4.2 Experimental design

All programs are measured on an SGI Origin2000 with R12K processors and an SGI O2 with a single R10K processor. The SGI O2 is included for a direct comparison with an earlier work by another group. Both the R12K and R10K provide hardware counters that measure cache misses and other hardware events with high accuracy. All machines have two caches: L1 uses 32-byte cache lines and is 32KB in size, L2 uses 128-byte cache lines, and the size of L2 is 1MB for O2 and 4MB for Origin2000. Both caches are two-way set associative. Both processors achieve good latency hiding as a result of dynamic, out-of-order instruction issue and compiler-directed prefetching. All applications are compiled with the highest optimization flag and prefetching on (f77 -n32 -mips4 -Ofast), except for *Sweep3D*, on which we use -O2 because it is 2% (23 seconds) faster than -Ofast for the original program (the performance improvement is similar at both optimization levels). The SGI compiler used is MIPSpro Version 7.30.

The five test applications are described in Table 1. They are benchmark programs from SPEC, NASA and DOE, except for *ADI*, which is a self-written program with separate loops processing boundary conditions. Since all programs use iterative algorithms, only the loops inside the time step were timed. However, the number of cache and TLB misses was measured for the entire execution.

We also examined other SPEC95fp applications, but failed to optimize them with the current compiler implementation because of two problems. The first is due to procedural abstraction in these programs where similar computations are represented by the same code but with different parameters. To fully expose the computation and data access, we need not only procedure inlining but an aggressive form of loop unrolling. We expect to extend our compiler implementation and optimize *tur3d*, *su2cor* and *hydro2d*. *Sweep3d* from DOE has the same problem, but we manually unrolled the outermost loops and then fused them. The second limitation is that the current implementation cannot transform programs where different layouts are used for the same region of memory, as in *mgrid*.

4.3 Effect of transformations

The effect of optimizations is shown in Figure 10. All results were collected on Origin2000 with R12K processors except for *Swim*, which was measured on SGI O2. Each graph shows three sets of bars: the original performance (normalized to 1), the effect of loop fusion, and the effect of loop fusion plus data regrouping. The figure also shows the execution time and original miss rate; however, comparisons are made on the number of misses, not on the miss rate.

Loop fusion and data grouping improved the two SPEC programs by 12% and 16%. *ADI* used a much larger data input than SPEC programs and saw a speedup of 2.33. The largest application, *Sweep3D*, was improved by a factor of 1.9 in overall performance. The rest of the section will consider *SP* in detail.

Program changes for *SP* SP is a full application benchmark and deserves special attention in evaluating the global strategy. The main computation subroutine, *adi*, uses 15 global data arrays in 218 loops, organized in 67 nests (after inlining). Loop distribution and loop unrolling result in 482 loops at three levels—157 loops at the first level, 161 at the second, and 164 at the third. Array splitting results in 42 arrays.

Loop fusion merges the whole program into 8 loops at the first level, 13 at the second and 17 at the third level. Data regrouping combines 42 arrays into 17 new ones. The choice of regrouping is very different from the initial arrays defined by the programmer. For example, the third new array consists of four original arrays: $\{ainv(N,N,N), us(N,N,N), qs(N,N,N), u(N,N,N,1-5)\}$, and the 15th new array includes two disjoint sections of an original array: $\{lhs(N,N,N,6-8), lhs(N,N,N,11-13)\}$.

Loop fusion eliminates almost a half of the L2 misses (49%). However, it packs too much data access into the

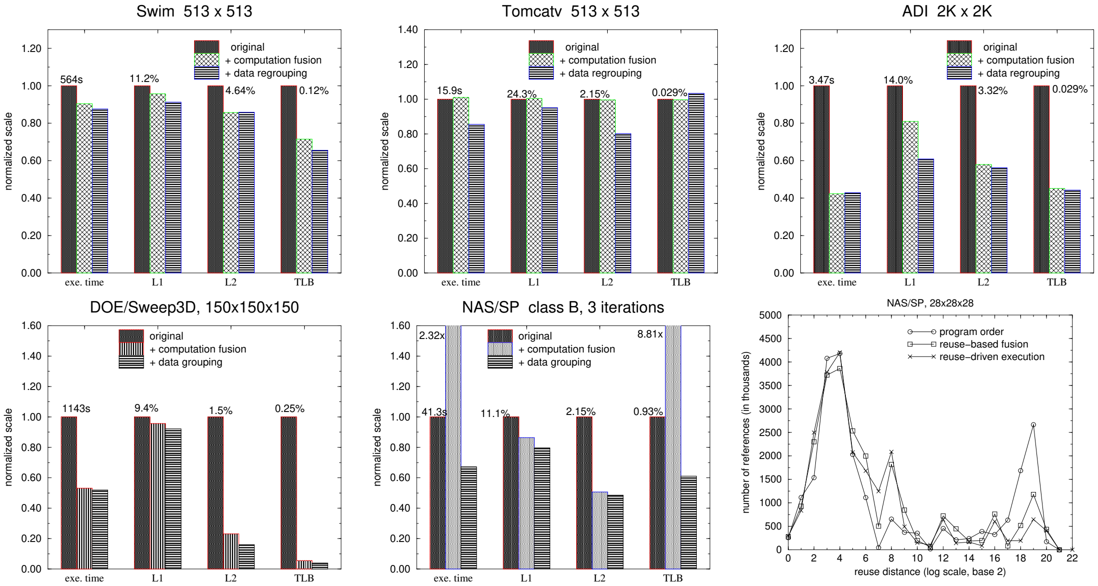


Figure 10. Effect of transformations

fused loop and causes 8 times more TLB misses, slowing the performance by a factor of 2.3. Data regrouping, however, merges related data in contiguous memory and reduces L1 misses by 20%, L2 by 51% and TLB by 39%. The execution time is shortened by one third (33%), a speedup of 1.5 (from 64.5 Mf/s to 96.2 Mf/s).

Next we compare the source-level loop fusion with trace-based reuse-driven execution, which was described in Section 2.1. As before, we use reuse distance to measure program locality. The lower rightmost graph in Figure 10 compares three versions of *SP*: the original program order, reuse-driven execution order, and the order after reuse-based loop fusion. Reuse-based fusion reduces the number of long reusd disntances by 45%, which is not as good as the 63% reduction by reuse-driven execution. However, loop fusion does realize a fairly large portion of the potential. Furthermore, the reduction on long reuse distances (45%) is very close to the reduction of L2 misses on the Origin2000 (51%), indicating that the measurement of reuse distance closely matches L2 cache performance on that machine.

4.4 Summary

The combined strategy of loop fusion and data regrouping is extremely effective for the benchmark programs tested, improving overall speed by between 14% and a factor of 2.33 for kernels and factors of 1.5 and 1.9 for the two full applications. The improvement can be obtained solely through automatic source-to-source compiler optimization. The success especially underscores the following three important contributions:

- Aggressive loop fusion. Reuse-based fusion can fuse loops of different shapes and therefore find more fusion opportunities. In *NAS/SP*, over one hundred loops were fused into a single loop.
- Conservative multi-level data regrouping. Data regrouping improves performance in all but one case, which can be corrected if the data transformation is applied by the back-end compiler. Since most fused loops are imperfectly nested, multi-level regrouping is necessary to maximize spatial locality. For example, *tomcatv* sees no chance of single-level regrouping, but multi-level regrouping improves performance by 16%.
- Combining strategies. When individually applied, loop fusion may significantly degrade performance without data regrouping, and data regrouping may see little opportunity without loop fusion[3]. *It is the combination that gives us a consistently effective strategy for global optimization.*

5 Related work

Many researchers have studied loop fusion. Early work was carried out by Wolfe[22], and Allen and Kennedy[3]. Combining loop fusion with distribution was originally discussed by Allen et al [2]. To improve reuse of vector registers, Allen and Kennedy fused loops with identical bounds, no fusion-preventing dependences, and no true dependences with intervening statements. Callahan developed a greedy fusion that maximizes coarse-grain parallelism [7]. His method may fuse loops of no data sharing.

The importance of global data reuse was shown by recent simulation studies. McKinley and Temam observed

[3] In *NAS/SP*, for example, data regrouping merged only two arrays in the absence of loop fusion, and improved performance by 8%. We tested data regrouping without loop fusion and reported the results in Ding's dissertation [9].

that most misses in SPEC and Perfect benchmarks are due to inter-nest temporal reuse [18]. Our reuse-driven execution complements their results by showing how much of the inter-nest reuse can be converted to cache reuse by program reordering. Although we used a particular heuristic, similar measurement can be used to evaluate other program reordering schemes.

Carr et al. implemented loop fusion for cache reuse [8]. They followed the same fusion constraints as Allen and Kennedy and fused on average 6% of tested loops. Fusion-preventing dependences can be avoided by *peel-and-jam*, introduced by Porterfield [19]. Peel-and-jam is a limited form of our loop alignment because peel-and-jam can only shift the first loop up (or the second loop down), but not the reverse. So it does not always bring together data reuses in fused loops. Loop alignment was originally used by Allen et al. to assist parallelization [2]. Manjikian and Abdelrahman used *peel-and-jam* to fuse loops possibly with different iteration bounds but with the same shape across the fused levels [17]. They found more opportunities for fusion. Still, at most 8 original loops could be fused into a single loop. Our fusion algorithm fused loops of different shapes and achieved higher degree of fusion. For example, about 500 loops in *NAS/SP* were fused into 8 loop nests.

Global loop fusion can be formulated as a graph-partitioning problem, which was studied independently by Gao et al. [13] and by Kennedy and McKinley [15]. They model loops as nodes and data reuse as weighted edges between pairs of loop nodes. Ding and Kennedy extended the formulation to base it on hyper-graphs where an edge (data sharing) connects an arbitrary number of loop nodes [11]. Recently, Kennedy developed a fast algorithm that always fuses along the heaviest edge and supports both models of reuse [14]. Graph-based methods have more freedom than the sequential scheme used in our work. Regardless of the heuristic used, global fusion must address three problems: how to fuse loops of different shapes, how to optimize data layout after fusion, and how to bound the potentially exponential cost of legality test. The solutions developed by this work—fusion-enabling transformations, data regrouping, and pair-wise fusion—are equally applicable to other fusion methods.

Some recent techniques reorganize loops completely based on their data access. Kodukula et al. blocked data and "shackled" computations on each data tile [16]. Similarly, Pugh and Rosser sliced loop iterations on each data element or data block [21]. Yi et al. improved iteration slicing and used it to convert loops into recursive functions [23]. Since these methods reorganize runtime instances of loop statements, they can fuse loops of different shapes at the same granularity as loop distribution and iteration reordering used in our work. However, questions remain on how to choose between fusion candidates, efficiently compute all-to-all transitive dependence, and optimize data layout after fusion. Although the above techniques have been successfully used to block single loop nests or small programs, they have not been tried on programs with a large number of (possibly not all fusible) loops. Pugh and Rosser tested iteration slicing on programs of multiple loop nests and found mixed results. On SGI Octane, *Swim* was improved by 10% but the transformation on *Tomcatv* "interacted poorly with the SGI compiler" [21].

Loop fusion may cause poor spatial locality due to the increased data access in fused loops. McKinley et al. reported that fusion improved hit rate in four programs but reduced the performance for another three programs [8]. Manjikian and Abdelrahman used a form of array padding (called cache partitioning) and found satisfactory result [17]. Array padding is a restricted form of data transformation because it reorganizes data only at coarse granularity. In addition, Manjikian and Abdelrahman did not address the problem where different loops require different data layouts. In comparison, array regrouping combines data at fine granularity with guaranteed profitability for the whole program [10]. In this work, we extend the regrouping to data of multiple granularity and use it in conjunction with loop fusion.

We are not aware of any published work that performs global computation fusion and data grouping either by hand or by hardware or operating systems. Such transformations should not be attempted by a programmer. They conflict with the modularity of the program because the proper data layout depends on computation organization. Hardware or operating systems are not well suited either because they have limited scope and incur runtime overhead.

6 Conclusion

This work has developed a global compiler strategy to alleviate the bandwidth limitations of modern machines by improving reuse of data from cache. The strategy includes reuse-based loop fusion and multi-level data regrouping. The former distributes and re-fuses all loops of different shapes; the latter splits and regroups all arrays at multiple granularity. Together they produce to date the most aggressive form of global yet fine-grained strategy for improving global cache reuse in large applications.

The implementation and evaluation have verified that the new global strategy can achieve dramatic reductions in the volume of data transferred for the programs studied. The table in Table 2 compares the amount of data transferred for versions of each program with no optimization, with optimizations provided by the SGI compiler, and after transformation via the strategy developed in this paper. If we compare the average *reduction* in misses due to compiler techniques, the new strategy, labeled by column *New*, does better than the SGI compiler by factors of 8 for L1 misses, 5 for L2 misses, 2.5 for TLB misses, and 1.6 for perfor-

program	L1 misses			L2 misses			TLB misses			Speedup
	NoOpt	**SGI**	**New**	**NoOpt**	**SGI**	**New**	**NoOpt**	**SGI**	**New**	over SGI
Swim	1.00	1.26	1.15	1.00	1.10	0.94	1.00	1.60	1.05	1.14
Tomcatv	1.00	1.02	0.97	1.00	0.49	0.39	1.00	0.010	0.010	1.17
ADI	1.00	0.66	0.40	1.00	0.94	0.53	1.00	0.011	0.005	2.33
DOE/Sweep	1.00	1.00	0.92	1.00	0.99	0.16	1.00	1.00	0.04	1.93
NAS/SP	1.00	0.97	0.77	1.00	1.00	0.49	1.00	1.09	0.67	1.49
average	**1.00**	**0.98**	**0.84**	**1.00**	**0.90**	**0.50**	**1.00**	**0.74**	**0.35**	**1.61**

Table 2. Overall Comparison

mance. In addition, the figures of the last two applications show that SGI compiler is not very effective in optimizing the two large applications, while the new strategy works especially well. Thus, the global strategy we propose has a clear advantage over the more local strategies employed by an excellent commercial compiler, especially for large programs with huge data sets.

Acknowledgment The implementation of this work is based on the D System of Rice University. We also thank anonymous referees for helpful comments on the early drafts of this paper.

References

[1] V. Adve and J. Mellor-Crummey. Using Integer Sets for Data-Parallel Program Analysis and Optimization. In *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, Montreal, Canada, June 1998.

[2] J. R. Allen, D. Callahan, and K. Kennedy. Automatic decomposition of scientific programs for parallel execution. In *Proceedings of the Fourteenth Annual ACM Symposium on the Principles of Programming Languages*, Munich, Germany, Jan. 1987.

[3] J. R. Allen and K. Kennedy. Vector register allocation. *IEEE Transactions on Computers*, 41(10):1290–1317, Oct. 1992.

[4] R. Allen and K. Kennedy. *Advanced Compilation for High Performance Computers*. Morgan Kauffman. to be published October 2000.

[5] L. Belady. A study of replacment algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[6] D. C. Burger, J. R. Goodman, and A. Kagi. Memory bandwidth limitations of future microprocessors. In *Proceedings of the 23th International Symposium on Computer Architecture*, Philadelphia, PA, May 1996.

[7] D. Callahan. *A Global Approach to Detection of Parallelism*. PhD thesis, Dept. of Computer Science, Rice University, Mar. 1987.

[8] S. Carr, K. S. McKinley, and C.-W. Tseng. Compiler optimizations for improving data locality. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, San Jose, CA, Oct. 1994.

[9] C. Ding. *Improving Effective Bandwidth through Compiler Enhancement of Global and Dynamic Cache Reuse*. PhD thesis, Dept. of Computer Science, Rice University, January 2000.

[10] C. Ding and K. Kennedy. Inter-array data regrouping. In *Proceedings of The 12th International Workshop on Languages and Compilers for Parallel Computing*, La Jolla, California, August 1999.

[11] C. Ding and K. Kennedy. Memory bandwidth bottleneck and its amelioration by a compiler. In *Proceedings of 2000 International Parallel and Distribute Processing Symposium (IPDPS)*, Cancun, Mexico, May 2000.

[12] C. Ding and Y. Zhong. Reuse distance analysis. Technical Report UR-CS-TR-741, University of Rochester, Feburary 2001.

[13] G. Gao, R. Olsen, V. Sarkar, and R. Thekkath. Collective loop fusion for array contraction. In *Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing*, New Haven, CT, Aug. 1992.

[14] K. Kennedy. Fast greedy weighted fusion. In *Proceedings of the 2000 International Conference on Supercomputing*, Santa Fe, NM, May 2000.

[15] K. Kennedy and K. S. McKinley. Typed fusion with applications to parallel and sequential code generation. Technical Report TR93-208, Dept. of Computer Science, Rice University, Aug. 1993. (also available as CRPC-TR94370).

[16] I. Kodukula, N. Ahmed, and K. Pingali. Data-centric multi-level blocking. In *Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation*, Las Vegas, NV, June 1997.

[17] N. Manjikian and T. Abdelrahman. Fusion of loops for parallelism and locality. *IEEE Transactions on Parallel and Distributed Systems*, 8, 1997.

[18] K. S. McKinley and O. Temam. Quantifying loop nest locality using SPEC'95 and the perfect benchmarks. *ACM Transactions on Computer Systems*, 17(4):288–336, 1999.

[19] A. Porterfield. *Software Methods for Improvement of Cache Performance*. PhD thesis, Dept. of Computer Science, Rice University, May 1989.

[20] W. Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):102–114, Aug. 1992.

[21] W. Pugh and E. Rosser. Iteration space slicing for locality. In *Proceedings of the Twelfth Workshop on Languages and Compilers for Parallel Computing*, August 1999.

[22] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers*. PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, Oct. 1982.

[23] Q. Yi, V. Adve, and K. Kennedy. Transforming loops to recursion for multi-level memory hierarchies. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, Vancouver, Canada, June 2000.