Improving Effective Bandwidth through Compiler Enhancement of Global Cache Reuse

Chen Ding and Ken Kennedy

CRPC-TR99809-S
September 1999

Center for Research on Parallel Computation
Rice University
6100 South Main Street
CRPC - MS 41
Houston, TX 77005

Submitted November 1999

Improving Effective Bandwidth through Compiler Enhancement of Global Cache Reuse

Chen Ding Ken Kennedy
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 in loops. However, today's machines are increasingly limited by the insufficient memory bandwidth— latency-oriented techniques are inadequate because they do not seek to minimize the amount of data transferred from memory over the whole program. To address the bandwidth limitation, this paper explores the potential for global cache reuse—that is, reusing data across loops nests and over the entire program. In particular, the paper investigates a two-step 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 make them contiguous in memory. While the first step reduces the frequency of memory access, the second step improves its 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 long latencies to memory 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 during the same period[1]. Our own performance study found that scientific applications demand two to nine times more memory bandwidth than provided by a typical machine, and consequently, full applications can achieve only at most 10% to 16% of peak CPU performance on average[6]. This means that substantive performance gains can be made on most applications if only the memory bandwidth bottleneck can be alleviated.

To bridge the memory gap, modern machines provide high-bandwidth data caches in the hope that most memory accesses can be served by a buffered copy from cache instead of directly from memory. Although cache has been successful for programs with small working sets and simple access patterns, it is not effective for large, complex applications that repeatedly process a huge data set. Since large programs perform computation in many phases and access data in many different places, accesses to the same data item are typically separated by a wide gap and data access is often non-contiguous. If the reuse of a data item is separated by a long enough gap, the value may be evicted from cache before it is reused, causing unnecessary data transfer from memory. Non-contiguous access, on the other hand, wastes even existing cache resource by causing useless data to be transferred to cache. Thus, both waste memory bandwidth. Moreover, the extensive use of function and data abstraction aggravates the problem by fragmenting computations and data that could be otherwise cached together.

An optimizing compiler can alleviate the memory bandwidth problem by restructuring global computation and data to better utilize the cache hierarchy. However, existing compiler techniques do not do a complete job of solving this problem. For example, loop blocking—a standard technique for increasing reuse—is targeted for single loop nests and cannot exploit data reuse among disjoint loops. Loop fusion combines multiple loops, but existing fusion methods are too restrictive. They do not fuse loops of a different number of dimensions or loops with data dependences on the intervening statements. Neither do they systematically address the overhead of loop fusion. Consequently, previous studies showed that global program transformations like loop fusion did not improve performance by a significant degree and they sometimes caused a slowdown.

This paper presents techniques designed to minimize the overall bandwidth consumption of a program. This goal is achieved by using a two-step strategy: *first fuse computations on the same data and then group data used by the same*

[1]This estimate is based on historical data about CPU speed, memory pin count, and pin-bandwidth increases compiled by Burger et al[3].

computation. Our treatment begins with a study on the potential benefit of global computation fusion through data reuse analysis in large applications. The main focus is on two program transformations. The first is *reuse-based loop fusion*, which employs statement embedding, loop alignment, and iteration reordering to fuse loops whenever possible. Since aggressive loop fusion brings together a large amount of access to common data, the second transformation, *data regrouping*, splits and regroups global arrays to effect contiguous data access in the fused loop. The goal is to maximize global cache reuse including the temporal reuse of the global computation and the spatial reuse of the global data.

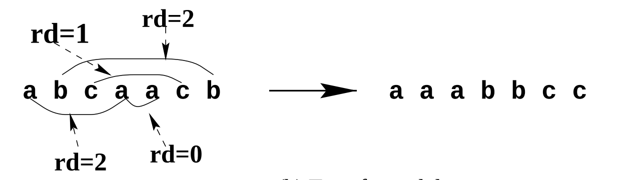
The rest of the paper is organized as follows. The next section studies data reuse behavior of programs and presents reuse-based loop fusion. Section 3 describes inter-array multi-level data regrouping. The implementation and evaluation of both techniques are presented in Section 4. Related work is discussed in Section 5 and the contributions of this work are summarized in Section 6.

2 Global computation fusion

This section first analyzes global data reuse in real programs and then presents the compiler algorithm for reuse-based loop fusion.

2.1 Reuse-distance analysis

In a sequential execution, the *reuse distance* of a data reference is the number of the *distinctive* data items appeared between this reference and the closest previous reference to the same data. The example in Figure 1(a) shows four data reuses and their reuse distance. On a perfect cache (fully associative with LRU replacement), a data reuse hits in cache if and only if its reuse distance is smaller than the cache size.



(a) Example sequence and its reuse distances

(b) Transformed data access sequence. All reuse distances are zero.

Figure 1: Example reuse distances

To avoid cache misses due to long reuse distances, a program can fuse computations on the same data. Figure 1(b) shows the computation sequence after fusion, where all reuse distances are reduced to zero. The next section studies fusion on real programs.

2.2 Reuse-driven execution

This section presents and evaluates *reuse-driven execution*, a machine-level strategy which fuses run-time instructions accessing the same data. In a sense, it is the inverse of Belady policy. While Belady evicts data that has the furthest reuse, reuse-driven execution executes the instruction that has the closest reuse. The insight gained in this study will provide the motivation for the source-level transformation presented in the next section.

Given a program, its reuse-driven execution is constructed as follows. First, the source program is instrumented to collect the run-time trace of all source-level instructions as well as all their data access. The trace is re-executed on an ideal parallel machine where an instruction is executed as soon as all its operands have been computed. The sequence of ideal execution gives the ordering of instructions and their minimal cycle difference. Finally, the reuse-driven execution is carried out by the algorithm given in Figure 2. It is reuse-driven because it gives priority of execution to later instructions that reuses the data of the current instruction. It employs a FIFO queue to sequentialize all preferred instructions.

The effect of reuse-driven execution is shown in Figure 3 for a kernel program *ADI* and a full application *NAS/SP*; 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 $[2^{(x-1)}, 2^x)$. The figure links discrete points into a curve to emphasize the elevated hills, where large portions of memory references reside. The important measure is not the length of a reuse distance, rather it is whether the length increases with the input size. If so, the memory access will become a cache miss when data input is sufficiently large. We call those reuses whose reuse distance increases with the input size *evadable reuses.*

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

Figure 2: Algorithm for reuse-driven execution

The upper two figures of Figure 3 show the reuse distances of *ADI* on two input sizes. The two curves in each figure show reuse distances of the original program and that of reuse-driven execution. In the original program, over 40% of memory references (25 thousand in the first and 99 thousand in the second) are evadable reuses. However, reuse-driven execution not only reduced the number of evadable reuses by 33% (from 40% to 27%), but also slowed the lengthening rate of the remaining evadable reuses.

A similar improvement is seen on *NAS/SP*, where reuse-driven execution reduced the number of evadable reuses by 63% and slowed the rate of lengthening of reuse distances. A third curve is shown in the lower-right figure, which will be explained later in the evaluation section. We also tested two other programs—a kernel, *FFT*, and a full application, *DOE/Sweep3D*. Reuse-driven execution did not improve *FFT* (where the number of evadable reuses was increased by 6%), but it reduced evadable reuses by 67% in *DOE/Sweep3D*. In addition, we experimented with other heuristics of reuse-driven execution, for example, that of not executing the next reuse if it is too far away (in the ideal parallel execution order). But the result was not improved. The experiment with reuse-driven execution demonstrates the potential of fusion as a global strategy for reducing the number of evadable reuses in large applications with multiple loop nests. The next section studies aggressive loop fusion as a way to realize this benefit.

2.3 Reuse-based loop fusion

Since loops contain most data access and data reuse, loop fusion is obviously a promising solution for shortening reuse distances. This section presents a new fusion algorithm that employs statement embedding, loop alignment and iteration reordering. It first demonstrates through an example the need for these three transformations in order to achieve maximal fusion and minimal reuse distance.

The fusion of the two loops shown in Figure 4(a) requires three steps: (1) embed s_3 into the first loop, (2) peel off the first iteration of the second loop, and (3) fuse the two loops by shifting the second loop up by two iterations. The first two steps make loop fusion possible, and the third step brings together the data reuse on $A[i]$. In programs with multi-dimensional loops, statement embedding is equivalent to inserting a low-dimensional loop as one iteration of a high-dimensional loop. Although statement embedding enables loop fusion in this example, it may fail to do so in certain cases, for example in the program shown in Figure 4(b). This complicates the test for the feasibility of loop fusion but suggests an advantage of incremental fusion, which fuses only consecutive loops.

The following discussion of loop fusion makes a number of assumptions on the input program, which are listed in Figure 5. At the beginning, we consider only single-dimensional loops accessing single-dimensional arrays. Later we will use the same algorithm to fuse multi-dimensional loops level by level. The other restrictions in Figure 5 can also be relaxed, however, at the cost of a more complex fusion algorithm. For example, index expressions like $A(d * i + c)$

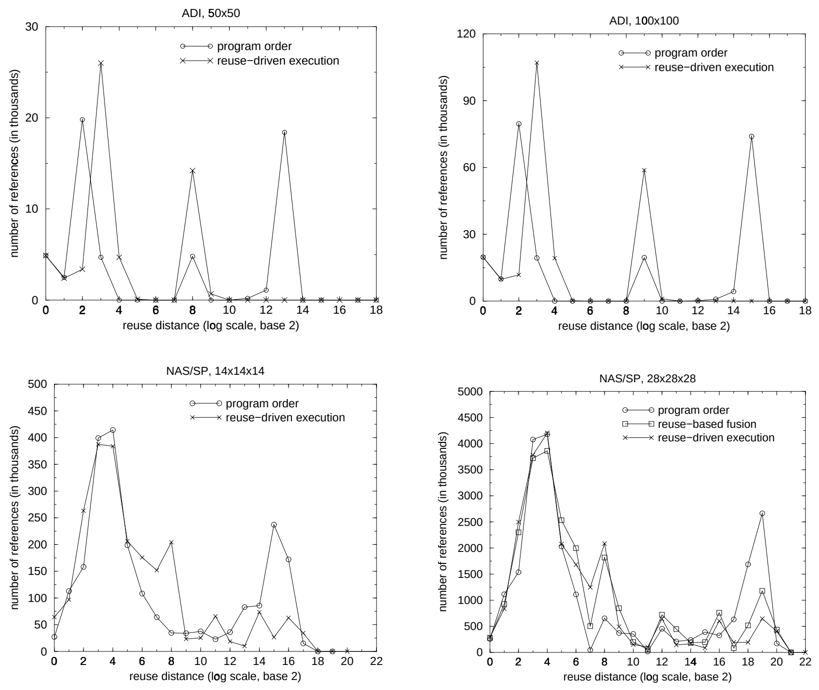


Figure 3: Effect of reuse-driven execution

```
    for i=3, N-2
      A[i] = f(A[i-1])
    end for
s3  A[1] = A[N]
s4  A[2] = 0.0
    for i=3, N
      B[i] = g(A[i-2])
    end for
```

$\longrightarrow$

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

A[1] = A[N]
B[3] = g(A[1])
```

(a) fusion by statement embedding, loop alignment and loop splitting

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

(b) example of loops that cannot be fused

Figure 4: Examples of loop fusion

can be considered by adding loop projection to loop fusion.

The fusion algorithm is given in Figure 6, which incrementally fuses all data-sharing loops. For each statement, subroutine *GreedilyFuse* tries to fuse it upwards with the closest predecessor (which is a fused loop) that accesses the same data. Once a fused loop is changed due to additional fusion, it will be re-tested for potential upward fusion because it now accesses a larger set of data. Subroutine *FusibleTest* determines whether two loops can be fused, and if so, what reordering is needed and what the minimal alignment factor is. The algorithm avoids repeated tests by remembering infusible loop pairs.

A fused loop is a collection of loops and non-loop statements, where loops are aligned with each other, and non-loop statements are embedded in some iteration of the fused loop. Its data footprint includes the access to all arrays. For each array, the data accessed is a loop-variant range such as $[i - c_1, i + c_2]$ plus a number of loop-invariant array locations. Data dependences are calculated by checking for non-empty intersections among footprints.

Three transformations can enable loop fusion. The first is statement embedding, which schedules a non-loop statement into a loop. The second is loop alignment, which shifts the second loop by k iterations, where k is called an *alignment factor*. The alignment factor can be negative if it is needed to bring together data reuses. When two loops cannot be fused by a bounded alignment factor, the third transformation, iteration reordering, can reshape and extract the fusible parts of loops. Examples of iteration reordering include loop splitting and loop reversal.

The three transformations achieve maximal fusion and minimal reuse distance. Statement embedding can always

- a program is a list of loops and non-loop statements
- all loops are one-dimensional and so are all variables
- all data accesses are in one of the two forms: $A[i+t]$ and $A[t]$, where A is the variable name, i is the loop index, and t is loop invariant

Figure 5: Assumptions on the input program

fuse a loop with a non-loop statement. Loop alignment and iteration reordering, on the other hand, may not always succeed in fusing two loops. However, when they fail, it must be the case that all iterations of the second loop depend on all iterations of the first loop. If so, loop fusion is impossible as a dependence-preserving transformation. Therefore, the above three transformations achieve maximal loop fusion. In addition to maximal fusion, the fusion algorithm achieves minimal reuse distance because all loops are fused with the minimal alignment factor.

After two loops are fused, their data reuses are no longer evadable, that is, their reuse distances remain unchanged when the input size grows. Excluding the access by non-loop statements, each iteration of the fused loop accesses $O(A)$ data elements, and the iteration difference between two data sharing statements is at most $O(L)$, where L is the number of loops in the program. Therefore, the upper bound on the distance of reuse is $O(A*L)$, which is independent of array sizes or data inputs. This upper bound is tight because a worst-case example can be constructed as follows: the first loop is `B(i)=A(i+1)`, then are L loops of `B(i)=B(i+1)`, finally is `A(i)=B(i)`. Since the two accesses to $A(i)$ must be separated by L iterations, the reuse distance can be no less than L. Therefore, the fusion algorithm achieves the tightest asymptotic upper bound on reuse distances.

The time complexity for the algorithm is $O(V * V' * (T + A))$, where V is the number of statements before fusion, V' is the number of loops after fusion, T is the cost of *FusibleTest*, and A is the number of program arrays. Our implementation made two simplifications. It assumed that all loop-invariant array access is on bordering elements, and it reordered iterations only by splitting at boundary loop iterations. These two assumptions were sufficient to capture all possible fusions in the programs we tested. The simplified fusion algorithm ran in $O(V * V' * A)$.

3 Global data regrouping

Although loop fusion shortens data reuse in computations, it does not optimize data layout. Indeed, a fused loop often touches a large number of arrays. The scattered data access may waste a major portion within a cache block and cause serious interference among cache blocks: both lead to low utilization of cache resources.

This section presents *data regrouping*, which clusters data used by the same computation to effect contiguous data access in a fused loop. To change the global data layout, data regrouping must solve two problems. The first is to analyze data structures of different sizes and different access patterns. The second, more difficult one, is to avoid undesirable side effects of a data transformation.

The program analysis for data regrouping has been reported in an earlier workshop paper of ours[5], so has the algorithm for array regrouping on a single data dimension. Data regrouping first partitions a program into a sequence of computation phases, each of which accesses data that is larger than cache. Then, it classifies all data arrays into compatible groups. Two arrays are compatible if and only if their sizes differ by at most a constant factor, and they are always accessed in the same order. Any data dimension of a small constant size is unrolled and 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.* Data regrouping uses an efficient partitioning algorithm to maximize the size of each partition. The profitability is guaranteed because no useless data is introduced into cache blocks by data regrouping. When useless data and dynamic regrouping are disallowed, the regrouping algorithm is optimal. The relaxation of either constraints makes optimal data layout dependent on the exact run-time tradeoff of a data layout. In contrast, conservative data regrouping achieves the best machine-independent data layout, that is, compile-time optimal solution. Relaxing either constraints also leads to NP-hard problems.

3.1 Multi-level regrouping

Our previous work was aimed at improving cache-block reuse and therefore did not group data at a granularity larger than an array element. This section overcomes this limitation by grouping arrays at higher levels. The extension is beneficial because placing simultaneously used array segments reduces cache interference and the page-table working

```
Main
  let p be the list of program statements, either loops or non-loop statements
  iterate p[i] from the first statement to the last in the program
     GreedilyFuse(p[i])
end Main

Subroutine GreedilyFuse(p[i])
  search backwards from p[i] to find the closest predecessor p[j] that shares data with p[i]
  do nothing and return to caller if p[j] does not exist
  if (p[i] is not a loop)
     embed p[i] into p[j]
     make p[i] an empty statement
  else if (FusibleTest(p[i], p[j]) finds a constant alignment factor)
     if (no splitting is required)
        fuse p[i] into p[j] by aligning p[i] by the alignment factor
        make p[i] an empty statement
        GreedilyFuse(p[j])
     end if
     if (splitting is required)
        split p[i] and/or p[j] and fuse p[i] into p[j] by aligning p[i]
        make p[i] an empty statement
        GreedilyFuse(p[j])
        for each remaining pieces t’ after splitting
            GreedilyFuse(t’)
     end if
  end if
end FuseStatement

Subroutine FusibleTest(p[i], p[j])
  if (p[i] p[j]) has been marked as not fusible
     return false
  end if
  for each array accessed in both p[i] and p[j]
     find the smallest alignment factor that satisfies data dependence and has the closest reuse
     apply iteration reordering if necessary and possible
  end for
  find the largest of all alignment factors
  if (the alignment factor is a bounded constant)
     return the alignment factor
  else
     mark (p[i] p[j]) not fusible
     return false
  end if
end FusibleTest
```

Figure 6: Algorithm for reuse-based loop fusion

set.

The example program in Figure 7 illustrates multi-level data regrouping. Array A and B are grouped at the element level to improve spatial reuse in cache blocks. In addition, the rows of all three arrays are grouped so that each outer loop iteration accesses a contiguous segment of memory. It should be noted that popular programming languages such as Fortran do not allow arrays of non-uniform dimensions like those of array D. However, this is not a problem when regrouping is applied by the back-end compiler.

```
for i
  for j
    g( A[j,i], B[j,i] )
  end for                          A[j,i] -> D[1,j,1,i]
  for j                            B[j,i] -> D[2,j,1,i]
    t( C[j,i] )                    C[j,i] ->   D[j,2,i]
  end for
end for
```

Figure 7: Example of multi-level data regrouping

The algorithm for multi-level regrouping is given in Figure 8. The first step of the algorithm disables data grouping at dimensions whose outer dimension is iterated by an inner loop. The second step iterates through all data dimensions and groups arrays at each dimension. The last step constructs the final layout by grouping dimension by dimension from outside to inside. The correctness of the third step can be established by proving that if two arrays are grouped at a given dimension, then both arrays are either fully grouped at all outer dimensions or not grouped at any outer data dimension.

4 Evaluation

4.1 Implementation

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 is array splitting and loop unrolling, which eliminates 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.

The data access of each loop is summarized by its data footprint. For each dimension of an array, a data footprint records 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.

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

For multi-level loops, loop fusion orders loop levels to maximize the benefit of fusion. The first loop level to fuse is the one that produces the fewest loops after fusion. In our experiment, however, this was largely unnecessary as computations are mostly symmetric. One exception in our test cases was *Tomcatv*, where we performed level ordering (loop interchange) by hand.

The analysis for data regrouping is trivial with data footprints. After fusion, data regrouping is applied level by level on the fused loops as specified by the algorithm given in Figure 8 but with two modifications. First, SGI compiler does a poor job when arrays are fully interleaved at the innermost data dimension. So we instead grouped arrays up to the second innermost dimension. This restriction may result in grouping in the less desired dimension, as in the case of *Tomcatv*. The other restriction is due to the limitation of Fortran language, which does not allow non-uniform array dimensions. In cases where multi-level regrouping produced non-uniform arrays, we changed it not to group at outer data dimensions.

Code generation is straightforward as mappings from old iteration space to the fused iteration space. Currently, the code is generated by the Omega library[14]. Omega worked well for small programs. where the compilation time

```
Assumptions
   all arrays have the same number of dimensions
   arrays are stored in column-major order

DataRegrouping
   for all memory access A(..., i, ..., j, ...)
       if (i loop is an outer loop of j loop)
          mark that A cannot be grouped at dimension j
   end for

   for each data dimension i from the highest to the lowest
       find all arrays that can be grouped at this dimension
       for each computation phase,
           find arrays that are accessed together at dimension i
       end for
       divide arrays into partitions where two arrays are in the same partition
           if and only if they are always accessed together
   end for

   for data dimension i from the highest to the lowest
       for each partition of dimension i found in the previous step
           group all arrays in the partition at data dimension i
       end for
   end for
```

Figure 8: Algorithm for multi-level data regrouping

was under one minute for all kernels. For *SP*, however, code generation took four minutes for one-level fusion and 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 three-level fusion. We are implementing a direct code generation scheme whose cost is linear to the number of loop levels and will report the result in the final paper.

4.2 Experimental design

We tested loop fusion and data regrouping on four applications described in Figure 9. The applications come from SPEC and NAS benchmark suite except *ADI*, which is a self-written kernel with separate loops processing boundary conditions. Since all programs use iterative algorithms, only the loops inside the time-step loop are counted. All programs are measured on a MIPS R12K processor of SGI Origin2000. A slower processor, R10K of SGI Octane is also used for a direct comparison with the earlier work by Pugh and Rosser[15]. Both R12K and R10K provide hardware counters that measure cache misses and other hardware events with high accuracy. Both machines have two caches: L1 is 32KB in size and uses 32-byte cache lines, L2 uses 128-byte cache lines, and the size of L2 is 1MB for Octane and 4MB for Origin2000. Both are two-way set associative. The MIPS processors achieve good latency hiding as a result of dynamic, out-of-order instruction issuing and compiler-directed prefetching. All applications are compiled with the highest optimization flag and prefetching turned on (f77 -mips4 -Ofast).

name	source	input size	No. lines	loop nests (levels)	No. arrays
Swim	SPEC95	513x513	429	6 (1-2)	15
Tomcatv	SPEC95	513x513	221	5 (1-2)	7
ADI	self-written	2Kx2K	108	6 (1-2)	3
SP	NAS/NPB Serial v2.3	class B, 3 iterations	1141	67 (2-4)	15

Figure 9: Applications tested

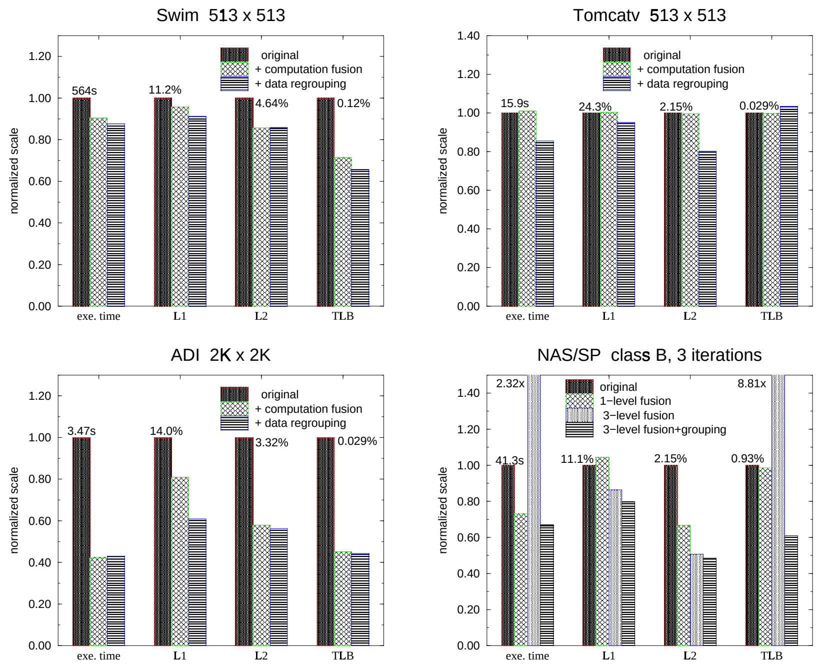


Figure 10: Effect of Transformations

4.3 Effect of transformations

The effect of optimizations on the execution time and the number of L1, L2 and TLB misses are shown in Figure 10. All results were collected on Origin2000 except *Swim*, which was on Octane. The first three applications show three sets of bars: the original performance (normalized to 1), the effect of loop fusion, and the effect of loop fusion plus data regrouping. For *SP*, one additional set of bars is used to show the effect of fusing one loop level instead of three loop levels. The execution time and original miss rates are also given in the figures; however, reductions are on the number of misses, not the miss rate.

The performance of *Swim* is reported for Octane because the same machine was used in the work of iteration slicing by Pugh and Rosser[15]. Reuse-based loop fusion achieved the same improvement (10%) as Pugh and Rosser reported for iteration slicing. The succeeding data grouping cut execution time by 2% more because of the additional reduction on L1 and TLB misses. On Origin2000, the number of L2 misses in the original program was reduced by 66% because of the larger L2 cache. The same loop fusion degraded performance by 6%, but the performance loss was recovered after data regrouping.

At the size 257x257 on Origin2000, *Tomcatv* exhibited similar behavior as *Swim*: loop fusion degraded performance by 2% but data regrouping recovered the loss (the combined transformation improved performance by 5% on Octane). At the larger size of 513x513, the performance gain of the combined strategy became apparent on Origin2000, as shown in Figure 10. Although loop fusion alone degraded performance by 1%, the combined transformation reduced L1 misses by 5%, L2 misses by 20% and overall execution time by 16%. Data regrouping increased TLB misses by 3% because it grouped arrays at the outer data dimension to avoid the poor code generation of the SGI compiler (see Section 4.1).

ADI used the largest input size and consequently enjoyed the highest improvement. The reduction was 39% for L1 misses, 44% for L2 and 56% for TLB. The execution time was reduced by 57%, a speedup of 2.33.

Program changes for *SP* SP is a full application and deserves special attention in evaluating the global strategy. The main computation subroutine, *adi*, uses 15 global data arrays in 218 loops structured as 67 loop nests (after inlining). Loop distribution and loop unrolling resulted in 482 loops at three levels—157 loops at the first level, 161 at the second, and 164 at the third. One-level loop fusion merged 157 loops into 8. The performance is shown by the second bar in the lower-right figure of Figure 10. The full fusion further fused loops in the remaining two levels and produced 13 loops at the second level and 17 at the third. The performance of full fusion is shown by the third bar in the figure.

The fourth bar shows the effect of adding data regrouping to the three-level fusion. The original program has 15 global arrays. Array splitting resulted in 42 arrays. Data regrouping merged them into 17 new ones. The choice of regrouping is very different from the specification given by the programmer. For example, the third new array consists of the following 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 $\{lhs(N,N,N,6-8), lhs(N,N,N,11-13)\}$.

One-level fusion increased L1 misses by 5%, but reduced L2 misses by 33% and execution time by 27%, signaling that the original performance bottleneck was on memory bandwidth. Fusing all levels eliminated half of the L2 misses (49%). However, it created too much data access in the innermost loop and caused 8 times more TLB misses. The performance was slowed by a factor of 2.32. Data regrouping, however, merged related data together and achieved the best performance. It reduced L1 misses by 20%, L2 by 51% and TLB by 39%. The execution time was shortened by one third (33%), a speedup of 1.5 (from 64.5 Mf/s to 96.2 Mf/s).

The reduction of L2 misses on Origin2000 (51%) is close to that of the simulation, which is 45% reduction on evadable reuses in Figure 3. Although reuse-based fusion at the source level is not as good as the ideal reuse-driven execution (which reduces the number of evadable reuses by 63%), it realizes a fairly large portion of its potential.

Summary The combined transformation of loop fusion and data regrouping are extremely effective for the benchmark programs tested, improving overall speed by 14% to a factor of 2.33 for kernels and a factor of 1.5 for the full application; furthermore, the improvement is obtained solely through automatic source-to-source compiler optimization. The success especially underscores the importance of the following two aspects of this work.

- Aggressive loop fusion. All test programs have loops with a different number of dimensions. Mere loop alignment cannot fuse any of the tested programs except for a few loops in *SP*. *Swim* also requires loop splitting.
- Combined optimization strategy of computation fusion and data regrouping. Although both together are always beneficial, neither of them is so without the other. Fusion may degrade performance without grouping and grouping may see little opportunity without fusion.

5 Related work

Many researchers have studied loop fusion. Early work includes those of Abu-Sufah et al[1], Wolfe[17], and Allen and Kennedy[2]. They restricted fusion to loops with identical bounds, no fusion-preventing dependences, and no true dependences with intervening statements. In addition, they considered only a local list of loops. Callahan employed greedy fusion as a global strategy for detecting coarse-grain parallelism[4]. His method, however, may fuse loops of no data sharing.

Porterfield introduced a transformation called *peel-and-jam*, which can fuse loops with fusion-preventing dependences by peeling off some iterations of the first loop and then applying fusion on the remaining parts[13]. While Porterfield considered only a pair of loops, Manjikian and Abdelrahman later extended peel-and-jam to find the minimal peeling factor for a group of fusible loops[11]. Also enabled by peel-and-jam, Song and Li developed a new tiling method that blocks multiple loops within a time-step loop[16]. However, these methods are not a complete global strategy because they did not address the cases where not all loops in a program are fusible. In addition, peel-and-jam is a limited form of loop alignment because it 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 the fused loops. Finally, peel-and-jam cannot fuse loops that have intervening statements that use the same data.

To find a solution to global loop fusion, a graph partitioning formulation was studied independently both by Gao et al.[7] and by Kennedy and McKinley[9]. Their aim was to improve temporal reuse in registers, and they model the benefit of register reuse as weighted edges between pairs of loops. The goal was to partition all loops into legal fusible groups so that the inter-group edge weight (unrealized data reuse) is minimal. Kennedy and McKinley proved that the general fusion problem is NP-Complete. Both studies used the heuristic that recursively applies min-cut algorithm to bi-partition the graph. Both work avoided fusing loops with fusion-preventing dependences. Since a weighted-edge between two loops does not truly model data sharing and data reuse, Ding and Kennedy extended the formulation to base it on partitioning hyper-graphs where an edge (data sharing) connects an arbitrary number of nodes[6]. Contemporarily with this work, Kennedy developed a fast algorithm that always fuse along the heaviest edge[8]. His algorithm allows accurate modeling of data sharing as well as the use of fusion enabling transformations. But none of these algorithms has been implemented or evaluated.

The first implementation for fusion and its evaluation on non-trivial programs were accomplished by McKinley et al[12]. They fused only loops with an equal number of iterations and with no fusion-preventing dependences. As a result, only 80 out of 1400, or 6% of tested loops were fused. They fused two loops in *Tomcatv* but found no improvement; they fused four loops in *SP* and obtained 0.24% increase in hit rate; they did not fuse any loop in *Swim*. The effect on full applications was mixed: fusion improved the hit rate for four out of 35 programs by 0.24% to 0.95%, but it also degraded performance of other three programs.

To enable more aggressive loop fusion, some researchers have taken a radically different approach. Instead of blocking loops, Kodukula et al. tiled data and "shackled" computations on each data tile[10]. Similarly, Pugh and Rosser sliced computations on each data element or data block[15]. Although effective for blocking single loops, data-oriented approaches are not practical as a global strategy for three reasons. First, without regular loop structures, it is not clear how to formulate and direct a global transformation. The shape of the transformed program is highly dependent on the choice of not only the shackled or sliced data but also of its starting loop. Furthermore, to maintain correctness, these methods need to compute all-to-all transitive dependences, whose complexity is cubical to the number of memory references in a program. Even when the dependence information is available, it is still not clear how to derive the best partitioning and ordering of the computations on different data elements, especially in the face of a large amount of unstructured computation. Finally, it is not clear how data-oriented transformations interacts with traditional loop-based transformations, and how the side effect of fusion can be tackled by either data regrouping or other methods. Kodukula et al. did not apply their work beyond a single loop nest[10]. Pugh and Rosser tested *Swim* and *Tomcatv* and found mixed results. On SGI Octane, the former was improved by 10% but the latter "interacted poorly with the SGI compiler"[15].

The previous work on loop fusion did not combine it with data transformations with two exceptions. Manjikian and Abdelrahman, who applied padding to reduce cache conflicts[11]. Array padding is less effective than inter-array spatial reuse because the latter eliminates cache conflicts by placing simultaneously used data into the same cache block. In fact, SGI compiler has padding as a part of its optimization but still causes serious fusion overhead. In a recent study, Ding and Kennedy used array shrinking and peeling to perform write-back reduction after loop fusion and before data regrouping[6]. Neither work has been tested on non-trivial programs.

Data regrouping is related to many data transformations, which have been discussed in [5]. Among those, data regrouping is the first one to selectively combine multiple arrays with guaranteed profitability and static optimality. And in this work, data regrouping is extended to grouping at multiple levels of high-dimensional data.

6 Contributions

This work has developed a global compiler strategy designed to alleviate the bandwidth limitations of modern machines by improving reuse of data from cache. The strategy employs a combination of two different transformations. *Reuse-based loop fusion* maximizes fusion and minimizes reuse distance with the help of statement embedding, loop alignment and iteration reordering. *Multi-level data regrouping* selectively regroups global arrays at multiple levels with guaranteed profitability and static optimality.

The implementation and evaluation have verified that the new global strategy achieved dramatic reductions in the volume of data transferred for the programs studied. The following table 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 9 for L1 misses, 3.4 for L2 misses, and 1.8 for TLB misses. Thus, the global strategy we propose has a clear advantage over the more local strategies employed by an excellent commercial compiler.

program	L1 misses			L2 misses			TLB misses		
	NoOpt	**SGI**	**New**	**NoOpt**	**SGI**	**New**	**NoOpt**	**SGI**	**New**
Swim	1.00	1.26	1.15	1.00	1.10	0.94	1.00	1.60	1.05
Tomcatv	1.00	1.02	0.97	1.00	0.49	0.39	1.00	0.010	0.010
ADI	1.00	0.66	0.40	1.00	0.94	0.53	1.00	0.011	0.005
NAS/SP	1.00	0.97	0.77	1.00	1.00	0.49	1.00	1.09	0.67
average	**1.00**	**0.98**	**0.82**	**1.00**	**0.88**	**0.59**	**1.00**	**0.68**	**0.43**

References

[1] W. Abu-Sufah, D. Kuck, and D. Lawrie. On the performance enhancement of paging systems through program analysis and transformations. *IEEE Transactions on Computers*, C-30(5):341–356, May 1981.

[2] J. R. Allen. *Dependence Analysis for Subscripted Variables and Its Application to Program Transformations*. PhD thesis, Dept. of Computer Science, Rice University, April 1983.

[3] 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*, 1996.

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

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

[6] C. Ding and K. Kennedy. Memory bandwidth bottleneck and its amelioration by a compiler. Technical report, Rice University, May 1999. Submitted for publication.

[7] 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, August 1992.

[8] K. Kennedy. Fast greedy weighted fusion. Technical Report CRPC-TR-99789, Center for Research on Parallel Computation (CRPC), 1999.

[9] 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, August 1993. (also available as CRPC-TR94370).

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

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

[12] 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.

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

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

[15] 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.

[16] Y. Song and Z. Li. New tiling techniques to improve cache temporal locality. In *ACM SIGPLAN Conference on Programming Languages Design and Implementation*, 1999.

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