Improving Cache Performance in Dynamic Applications through Data and Computation Reorganization at Run Time*

Chen Ding Ken Kennedy

{cding, ken}@cs.rice.edu
Computer Science Department
Rice University
Houston, TX 77005

Abstract

With the rapid improvement of processor speed, performance of the memory hierarchy has become the principal bottleneck for most applications. A number of compiler transformations have been developed to improve data reuse in cache and registers, thus reducing the total number of direct memory accesses in a program. Until now, however, most data reuse transformations have been *static*—applied only at compile time. As a result, these transformations cannot be used to optimize irregular and dynamic applications, in which the data layout and data access patterns remain unknown until run time and may even change during the computation.

In this paper, we explore ways to achieve better data reuse in irregular and dynamic applications by building on the inspector-executor method used by Saltz for run-time parallelization. In particular, we present and evaluate a *dynamic* approach for improving both computation and data locality in irregular programs. Our results demonstrate that run-time program transformations can substantially improve computation and data locality and, despite the complexity and cost involved, a compiler can automate such transformations, eliminating much of the associated run-time overhead.

1 Introduction

As modern single-chip processors have increased the rate at which they execute instructions, performance of the memory hierarchy has become the bottleneck for most applications. In the past, the principal challenge in memory hierarchy management has been overcoming latency, but blocking and prefetching have ameliorated that problem significantly. As exposed memory latency is reduced, bandwidth has become the dominant performance constraint because limited memory bandwidth bounds the rate of data transfer between memory and CPU regardless of the speed of processors or the latency of memory access. Our experiments on the SGI Origin 2000 have indicated that the bandwidth needed to achieve peak performance levels on most scientific applications on large data sets is a factor of two or more greater than that provided by the memory system[11]. As a result, program performance is now limited by its effective bandwidth, that is, the rate at which operands of a computation are transferred between CPU and memory.

Currently, the principal software mechanism for improving effective bandwidth in a program, as well as reducing overall memory latency, is increasing temporal and spatial reuse through program transformation. *Temporal reuse* occurs when multiple accesses to the same data structure use a buffered copy in cache or registers, eliminating the need for repeated accesses to main memory. While temporal reuse reduces the frequency of memory accesses, *spatial reuse* improves the efficiency of each memory access by grouping accesses on the same cache line. Since most current machines transfer one cache line at a time from memory, this grouping amortizes the cost of the bandwidth over more references. The combination of temporal and spatial reuse can minimize the number of transferred cache lines, i.e. the total memory bandwidth requirement of the program.

A substantive portion of the research on compiler memory management has focused on increasing temporal and spatial reuse in regular applications. Cache and register blocking techniques group computations on data tiles to enhance temporal reuse[5, 22]. Various loop reordering schemes seek to arrange stride-one data access to maximize spatial reuse[1, 12, 17]. Data transformations can often be used to effect spatial reuse when computation transformation is insufficient or illegal[8].

*Accepted for publication in ACM SIGPLAN PLDI'99.

None of these strategies, however, works well with dynamic and irregular computations because the unpredictable nature of data reuse prevents effective static analysis. An example is molecular dynamics simulation, which models the movement of particles in some physical domain (e.g. a 3-D space). The distribution of molecules remains unknown until run time, and the distribution itself changes during the computation. Another class of dynamic applications employ sparse linear algebra, where the non-zero entries in a sparse matrix changes dynamically. In both types of computation, it is impossible to enhance dynamic temporal reuse and irregular spatial reuse with static transformations.

The alternative to static methods is to apply dynamic reorganization at run time. Such strategies have been routinely employed to enhance the efficiency of parallel computations using the so-called "inspector-executor" method pioneered by Saltz and his colleagues. The underlying strategy is to insert code into the object program that reorganizes the computation or data layout once the structure of that data is known. The cost of this reorganization is then amortized over numerous time steps of the computation[10].

In this paper we describe two run-time transformations that improve the memory hierarchy performance of irregular computations like molecular dynamics simulation, and we present experimental evidence of their effectiveness. The *locality grouping* transformation reorders computation to improve dynamic temporal reuse. *Dynamic data packing*, on the other hand, reorganizes data to achieve better spatial locality. In addition to these two transformations, we discuss a static data transformation, *data regrouping*, that is necessary to optimize global static data layout for large dynamic programs.

A substantial portion of this paper is devoted to the compiler support for dynamic data packing. Transforming data at run time carries a significant overhead because of the need to redirect accesses from the old layout to the transformed one. However, most of this overhead can be eliminated by compiler optimizations. This paper describes an implementation of packing and evaluates the associated optimizations.

The remainder of the paper is organized as follows. Section 2 describes locality grouping and dynamic data packing, with a simulation study on their effectiveness on various cache configurations. Section 3 presents the compiler support for dynamic data packing, including optimizations that eliminate most of its run-time overhead. Section 4 briefly discusses data regrouping. In Section 5, the three transformations are evaluated on the SGI Origin2000 using three well-known benchmarks and a full application. Related work is discussed in Section 6. Finally, Section 7 summarizes the original contributions of this paper.

2 Run-time Computation and Data Transformations

This section describes two run-time transformations: locality grouping, which reorders data access to improve dynamic temporal reuse; and dynamic data packing, which reorganizes data layout for better run-time spatial reuse. Both transformations are then evaluated, individually and combined, through various access sequences on simulated caches.

2.1 Locality Grouping

The effectiveness of cache is predicated on the existence of locality and good computation structure exploiting that locality. In a dynamic application such as molecular dynamics simulation, the locality comes directly from its physical model in which a particle interacts only with its neighbors. A set of neighboring particles forms a locality group in which most interactions occur within the group. In most programs, however, locality groups are not well separated. Although schemes such as domain partitioning exist for explicitly extracting locality, they are very time-consuming and may therefore not be cost-effective in improving cache performance of a sequential execution. To pursue a better tradeoff, this section proposes the most efficient, yet also very powerful reordering scheme, locality grouping.

Given a sequence of objects and their interactions, locality grouping goes through the list of objects and clusters all interactions involving each object in the list. Figure 1 shows an example of locality grouping. Graph (a) draws the example objects and their interactions and Graph (b) is an example enumeration of all interactions. Assuming a cache of 3 objects, the example sequence incurs 10 misses. Locality grouping reorders the access sequence so that all interactions with each object are clustered. The new sequence then starts with all interactions on object a, then b, until the last object g. The locality-grouped access sequence incurs only 6 misses.

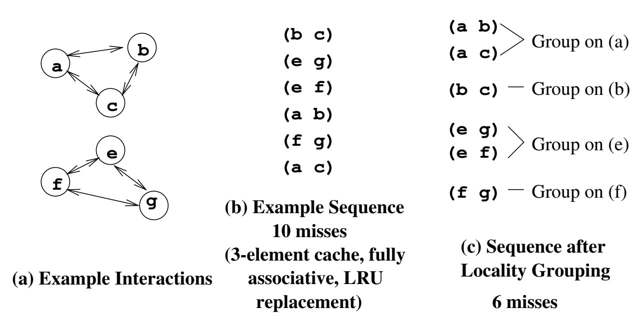

Figure 1: Example of Locality Grouping

Locality grouping incurs minimal run-time overhead. It can be done by doing a radix sort through two passes: the first pass collects a histogram and the second pass produces the locality-grouped sequence. Locality grouping also applies to interactions in tuples involving more than a pair of objects. A compiler can automate locality grouping by simply inserting a call to a sorting subroutine. The legality and profitability of this transformation can be determined either by compiler analysis or user directives, similar to the compiler support to run-time data transformations, which we will show in detail in the next section.

We evaluated locality grouping on a data set from *mesh*, a structural simulation. The data set is a list of edges of a mesh structure of some physical object such as an airplane. Each edge connects two nodes of the mesh. This specific data set, provided by the Chaos group at University of Maryland, has 10K nodes and 60K edges. We simulate only the data accesses on a fully associative cache in order to isolate the inherent cache reuse behavior from other factors. The two caches we simulate are 2K and 4K bytes in size and they use unit-length cache lines.

Table 1 gives the miss rate of *mesh* with and without locality grouping. Locality grouping eliminates 96.9% of cache misses in the 2K cache and 99.4% in the 4K cache. The miss rates after locality grouping are extremely low, especially in the 4K cache (0.37%). Further decreasing miss rate with more powerful reordering schemes in this case is unlikely to be cost-effective if the overhead of extra execution time does not out-weigh the additional gain.

	Original		After locality grouping	
	2K cache	4K	2K cache	4K
miss rate of *mesh*	93.2%	63.5%	2.93%	0.37%

Table 1: Effect of Locality Grouping

2.2 Dynamic Data Packing

Correct data placement is critical to effective use of available memory bandwidth. Placement of data elements in memory in the order in which they are accessed should improve spatial reuse. In regular computations, this placement can be done at compile time. However, in an irregular or adaptive computation, the order of data access is not known until run time and that order may change dynamically. *Dynamic data packing* is a run-time optimization that groups data accessed at close intervals in the program into the same cache line. For example, if two objects are always accessed consecutively in a computation, placing them adjacent to each other increases bandwidth utilization by increasing the number of bytes on each line that are used before the line is evicted.

Figure 2 will be used as an example throughout this section to illustrate the packing algorithms and their effects. Figure 2(a) shows an example access sequence. The objects are numbered by their location in memory. In the sequence, the first object interacts with the 600th and 800th object and subsequently the latter two objects interact with each other. Assume that the cache size is limited and the access to the last pair of the 600th and 800th objects cannot reuse the data loaded at the beginning. Since each of these three objects are on different cache lines, the total number of cache misses is 5. A transformed data layout is shown in Figure 2(b), where the three objects are relocated at positions 0 to 2. Assuming a cache line can hold three objects, the transformed layout only incurs two cache misses, a significant reduction from the previous figure of 5 misses.

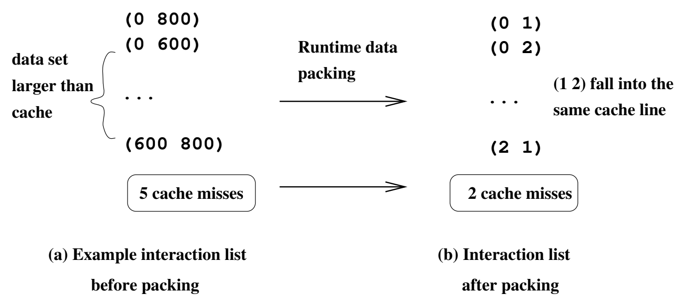

Figure 2: Example of Data Packing

The rest of this section presents three packing algorithms and a comparison study of their performance on different types of run-time inputs.

Packing Algorithms

The simplest packing strategy is to place data in the order they first appear in the access sequence. We call this strategy *consecutive packing* or *first-touch packing*. The packing algorithm is as follows. To ensure that each object has one and only one location in the new storage, the algorithm uses a tag for each object to label whether the object has been packed or not.

```
initializing each tag to be false (not packed)
for each object i in the access sequence
    if i has not been packed
       place i in the next available location
       mark its tag to be true (packed)
    end if
end iteration
place the remaining unpacked objects
```

Consecutive packing carries a minimal time and space overhead because it traverses the access sequence and object array once and only once. For access sequences in which each object is accessed at most once, consecutive packing yields optimal cache line utilization because the objects are visited in stride-one fashion during the computation. Achieving an optimal packing in the presence of repeated accesses, on the other hand, is NP-complete, as this problem can be reduced to the G-partition problem[16] following a similar reduction by Thabit[20]. The packing algorithms presented in this section are therefore based on heuristics.

One shortcoming of consecutive packing is that it does not take into account the different reuse patterns of different objects. *Group packing* attempts to overcome this problem by classifying objects according to their reuse pattern and applying consecutive packing within each group. In the example in Figure 2(b), the first object is not reused later but the 600th and 800th object are reused after a similar interval. Based on reuse patterns, group packing puts the latter two objects into a new group and packs them separately from the first object. If we assume a cache line of two objects, consecutive packing fails to put the latter two objects into one cache line but grouping packing succeeds. As a result, consecutive packing yields four misses while group packing incurs only three.

The key challenge for group packing is how to characterize a reuse pattern. The simplest approach is to use the average reappearance distance of each object in the access sequence, which can be efficiently computed in a single pass. More complex characterizations of reuse patterns may be desirable if a user or compiler has additional knowledge on how objects are reused. However, more complex reuse patterns may incur higher computation costs at run time.

The separation of objects based on reuse patterns is not always profitable. It is possible that two objects with the same reuse pattern are so far apart in the access sequence so that they can never be in cache simultaneously. In this case, we do not want to pack them together. To solve this problem, we need to consider the distance between objects in the access sequence as well as their reuse pattern. This consideration motivates the third packing algorithm, *consecutive-group packing*.

Consecutive-group packing groups objects based on the position of their first appearance. For example, it first groups the objects appeared in the first N positions in the access sequence, then the objects in the next N positions, and so on until the end of the access sequence. The parameter N is the *consecutive range*. Within each range group, objects can then be reorganized with group packing.

The length of the consecutive range determines the balance between exploiting closeness and exploiting reuse patterns. When the consecutive range is 1, data packing is the same as consecutive packing. When the range is the full sequence, the packing is the same as grouping packing. In this sense, these three packing algorithms are actually one single packing heuristic with different parameters.

Evaluation of Packing Algorithms

We evaluated all three packing algorithms on *mesh* and another input access stream that we extracted from *moldyn*, a molecular dynamics simulation program. The *Moldyn* program initializes approximately 8K molecules with random positions. As before, we simulated only the data access on a fully associative cache.

The group packing classifies objects by their average reappearance distance; it is parameterized by its distance granularity. A granularity of 1000 means that objects whose average reappearance distance fall in each 1000-element range are grouped together. Consecutive-group packing has two parameters: the first is the consecutive range, and the second is the grouping packing algorithm used inside each range.

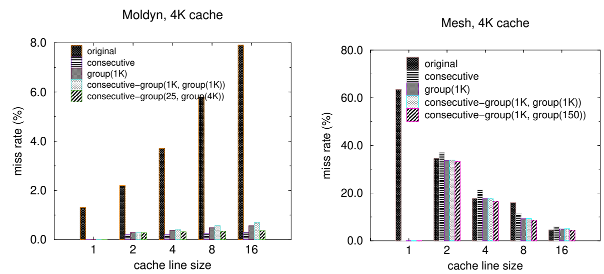

Figure 3: *moldyn* and *mesh*, 4K cache

The two graphs in Figure 3 show the effect of packing on the *moldyn* and *mesh* data sets. The left graph draws the miss rate on a 4K-sized cache for different cache line sizes from 1 to 16 molecules long. The miss rate of the original data layout, shown by the first bar of each cluster, increases dramatically as cache lines get longer. The cache with 16-molecule cache lines incurs 6 times the number of misses of the unit-line cache. Since the total amount of memory transfer is the number of misses times cache line size, the 16-molecule cache lines result in 96 times the memory transfer volume of the unit cache line case—it is wasting 99% of the available memory bandwidth! Even 2-molecule cache lines waste over 80% of available memory bandwidth. After various packing algorithms are applied, however, the miss rates drop significantly, as indicated in the remaining four bars in each cluster. Consecutive packing reduces the miss rate by factors ranging from 7.33 to over 26. Because of the absence of consistent reuse pattern, group and consecutive-group packing do not perform as well as consecutive packing but nevertheless reduce the miss rate by a similar amount.

The original access sequence of the *mesh* data set has a cyclic reuse pattern and a very high miss rate; see, for example, 64% on the 4K cache, shown in the right-hand graph of Figure 3. Interestingly, the cyclic data access pattern scales well on longer cache lines, except at the size of 8. Data packing, however, evenly reduces miss rate on all cache line sizes, including the size of 8. At that size, packing improves from 29% to 46%. On other sizes, consecutive packing and group packing yield slightly higher miss rates than the original data layout. One configuration, consecutive-group(1K,group(150)), is found to be the best of all; it achieves the lowest miss rate in all cases, although it is only marginally better on sizes other than 8. It should be noted that the result of consecutive-group packing is very close to the ideal case where the miss rate halves when cache line size doubles. As shown in the next section, dynamic packing, when combined with locality grouping, can reduce the miss rate to as low as 0.02%.

We also simulated 2K-sized caches and observed similar results. Consecutive packing reduces the miss rate of *moldyn* by 27% to a factor of 3.2. Consecutive-group packing improves *mesh* by 1% to 39%.

2.3 Combining Computation and Data Transformation

When we combine locality grouping with data packing on *mesh* (*moldyn* was already in locality-grouped form), the improvement is far greater than when they are individually applied. Figure 4 shows miss rates of *mesh* after locality grouping. On a 4K cache, the miss rate on a unit-line cache is reduced from 64% to 0.37% after locality grouping. On longer cache-line sizes, data packing further reduces the miss rate by 15% to a factor of over 6. On the 16-molecule cache line case, the combined effect is a reduction from a miss rate of 4.52% (shown in Figure 3) to 0.02%, a factor of 226. On a 2K cache with 16-molecule cache lines, the combined transformations reduce miss rate from 7.48% to 0.25%, a factor of 30. Although not shown in the graph, group and consecutive-group packing do not perform as well as consecutive packing.

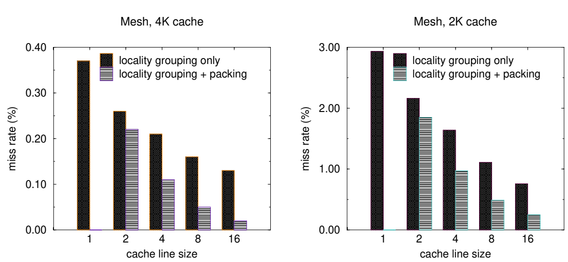

Figure 4: Mesh after Locality Grouping

In summary, the simulation results show that locality grouping effectively extracts computation locality, and data packing significantly improves data locality. The effect of data packing becomes even more pronounced in caches with longer cache lines. In both programs, simple consecutive packing performs the best after locality grouping, and the combination of locality grouping and consecutive packing yields the lowest miss rate.

3 Compiler Support for Dynamic Data Packing

Run-time data transformations, dynamic data packing in particular, involve redirecting memory accesses to each transformed data structure. Such run-time changes complicate program transformations and induce overhead during the execution. This section presents compiler strategies to automate data transformations and minimize their run-time overhead.

3.1 Packing and Packing Optimizations

The core mechanism for supporting packing is a run-time data map, which maps from the old location before data packing to the new location after data packing. Each access to a transformed array is augmented with the indirection of the corresponding run-time map. Thus the correctness of packing is ensured regardless the location and the frequency of packing. Some existing language features such as sequence and storage association in Fortran prevent a compiler from accurately detecting all accesses to a transformed array. However, this problem can be safely solved in a combination of compile, link and run-time checks described in [7].

Although the compiler support can guarantee the correctness of packing, it needs additional information to decide on the profitability of packing. Our compiler currently relies on a one-line user directive to specify whether packing should be applied, when and where packing should be carried out and which access sequence should be used to direct packing. The packing directive provides users with full power of controlling data packing, yet relieves them from any program transformation work. At the end of this section, we will show how the profitability analysis of packing can be automated without relying on any user-supplied directive.

The following example illustrates our compiler support for data packing. The example has two computation loops: the first loop calculates cumulative forces on each object, and the second loop calculates the new location of each object as a result of those forces. The packing directive specifies that packing is to be applied before the first loop.

```
Packing Directive: apply packing using interactions

for each pair (i,j) in interactions
   calculate_force( force[i], force[j] )
end for

for each object i
   update_location( location[i], force[i] )
end for
```

The straightforward (unoptimized) packing produces the following code. The call to *apply_packing* analyzes the *interactions* array, packs *force* array and generates the run-time data map, *inter$map*. After packing, indirections are added in both loops.

```
apply_packing( interactions[*], force[*], inter$map[*])
for each pair (i,j) in the interaction array
   calculate_force( force[ inter$map[i] ],
                    force[ inter$map[j] ] )
end for

for each object i
   update_location(location[i], force[ inter$map[i] ])
end for
```

The cost of data packing includes both data reorganization during packing and data redirection after packing. The first cost can be balanced by adjusting frequency of packing. Thus the cost of reorganizing data is amortized over multiple computation iterations. A compiler can make sure that this cost does not outweigh any performance gain by either applying packing infrequently or making it adjustable at run time. As will be shown in Section 5, data reorganization incurs negligible overhead in practice.

Data indirection, on the other hand, can be very expensive, because its cost is incurred on every access to a transformed array. The indirection overhead comes from two sources: the instruction overhead of indirection and the references to run-time data maps. The indirection instructions have a direct impact on the number of memory loads but the overhead becomes less significant in deeper memory hierarchy levels. However, the cost of run-time data maps has an consistent effect on all levels of cache, although this cost is likely to be small in cases where the same data map is shared by many data arrays. In addition, as we show next, the cost of indirection can be almost entirely eliminated by two compiler optimizations, *pointer update* and *array alignment*.

Pointer update modifies all references to transformed data arrays so that the indirections are no longer necessary. In the above example, this means that the references in *interactions* array are changed so that the indirections in the first loop can be completely eliminated. To implement this transformation correctly, a compiler must (1) make sure that every indirection array is associated with only one run-time data map and (2) when packing multiple times, maintain two maps for each run-time data map, one maps from the original layout and the other maps from the most recent data layout.

The indirections in the second loop can be eliminated by array alignment, which reorganizes the *location* array in the same way as the *force* array, that is, aligns the i's element of both arrays. Two requirements are necessary for this optimization to be legal: (1) the loop iterations can be arbitrarily reordered, and (2) the range of loop iterations is identical to the range of re-mapped data. The second optimization, strictly speaking, is more than a data transformation because it reorders loop iterations.

The following is the example code after applying pointer update and array alignment. The *update_map* array is added to map data from the last layout to the current layout. After the two transformations, all indirections through the *inter$map* array have been removed.

```
apply_packing( interactions[*], force[*],
               inter$map[*], update_map[*] )
update_indirection_array( interactions[*],
                          update_map[*] )
transform_data_array(location[*], update_map[*])

for each pair (i,j) in interactions
   calculate_force( force[i], force[j] )
end for

for each object i
   update_location( location[i], force[i] )
end for
```

The overhead of array alignment can be further reduced by avoiding packing those data arrays that are not live at the point of data packing. In the above example, if the *location* array does not carry any live values at the point of packing, then the third call, which transforms *location* array, can be removed.

3.2 Compiler Implementation

We have implemented the compiler support for packing in 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.

The first step of the compiler support is to find all possible packing candidates, and it does so by first discovering and then partitioning primitive packing groups. Each primitive packing group contains two sets of arrays: the set of access arrays, which hold the indirect access sequence, and the set of data arrays, which are either indirectly accessed through the first set of arrays or alignable with some arrays that are indirectly accessed. Given a program, the compiler identifies all premitive packing groups as follows. For each indirect data access in the program, the compiler puts the access array and the data array into a new primitive packing group. For each loop that can be freely reordered, the compiler puts all accessed data arrays into a new primitive packing group. Then the compiler partitions all primitive packing groups into disjoint packing partitions. Two primitive packing groups are disjoint if they don't share any array between their access array sets and between their data array sets. A union-find algorithm can efficiently perform the paritioning.

After partitioning, each disjoint packing partition is a possible packing candidate. The two optimizations can be readily applied to any packing candidate, should it become the choice of packing. Pointer update changes all arrays in the access array set; array alignment transforms all arrays in the data array set and reorders the loops that access aligned data arrays. For all other accesses that are not covered by the above two optimizations, the compiler inserts indirections through run-time maps.

The current implementation of packing has several limitations. It does not work on programs where the indirect access sequence is incrementally computed because the one-line directive requires the existence of a full access sequence. A possible extension would be to allow user to specify a region of computation in which to apply packing so that the compiler can record the full access sequence at run time. The other restriction of the current implementation is due to conservative handling of array parameter passing. For each subroutine with array parameters, we do not allow two different array layouts to be passed to the same formal parameter. This problem can be solved by propagating array layout information in a way similar to interprocedural constant propagation or data type analysis and then cloning the subroutine for each reachable array layout. In the programs we have encountered, however, there is no need for such cloning. The implementation also inherits limitations from our compiler infrastructure: it only compiles programs written in Fortran 77 and consequently it does not handle recursion. However, recursion should present no fundamental obstacles to these methods.

3.3 Extensions to Fully Automatic Packing

Although the one-line packing directive is convenient when a user knows how to apply packing, the mondatory requirement for such a directive is not desirable in situations when a user cannot make an accurate judgement on the profitability of packing. This section discusses several straightforward extentions which can fully automate the profitability analysis, specifically, extensions that decide whether, where, and when to apply packing.

With the algorithm described in the previous section, a compiler can identify all packing candidates. For each candidate, the compiler can record the access sequence at run time and determine whether it is noncontiguous and, if so, whether packing can improve its spatial reuse. Such decisions depend on program inputs and must be made with some sort of run-time feedback system. In addition, the same data may be indirectly accessed by more than one access sequence, each may demand a different reorganization scheme. Again, runtime analysis is necessary to pick out the best packing choice.

Once the compiler chooses a packing candidate, it can place packing calls right before the place where the indirect data accesses begin. The placement requires finding the right loop level under which the whole indirect access sequence is iterated.

The frequency of packing can also be automatically determined. One efficient scheme is to monitor the average data distance in an indirect access sequence and only invoke packing routines when adjacent computations access data that are too far apart in memory. Since the overhead of data reorganization can be easily monitored at run-time, the frequency of packing can be automatically controlled to balance the cost of data reorganization.

4 Optimal Data Regrouping

A dynamic application may have multiple computation phases each of which computes on a different but overlapping set of data. *Data regrouping* separates the data of different phases, computes the optimal grouping scheme and places data within each group consecutively in memory. The regrouping algorithm and a more detailed discussion can be found in [11]. In summary, optimal regrouping is equivalent to a set-partitioning problem and it can be computed in $O(min(max(2^S, N), N * \log N * S)$, where N is the number of arrays and S is the number of computation phases. For dynamic applications, optimal regrouping achieves full cache utilization and maximal spatial reuse. The regrouping algorithm also benefits regular applications because it guarantees full cache utilization and minimal working sets in cache and TLB. Existing compiler techniques are capable of implementing the analysis and transformation of data regrouping. In particular, bounded regular sections [14] can be used to analyze computation phases and to direct data transformations. It has been successfully applied for similar purposes in other contexts.

5 Evaluation

5.1 Experimental Design

Table 2 lists the four applications we used for the study, along with their description, source and size. We chose three scientific simulation applications from molecular dynamics, structural mechanics and hydrodynamics. Despite the difference in their physical model and computation, they have similar dynamic data access patterns in which objects interact with their neighbors. *Moldyn* and *mesh* are well-known benchmarks. We used a large input data set for *moldyn* with random initialization. *Mesh* has a user-supplied input set. *Magi* is a full, real-world application consisting of almost 10,000 lines of Fortran 90 code. In addition to the three simulation programs, we included a sparse-matrix benchmark to show the effect of packing on irregular data accesses in such applications.

The test programs are measured on a single MIPS R10K processor of an SGI Origin2000. The R10K provides hardware counters that measure cache misses and other hardware events with a very small run-time overhead. The processor has two caches: the first-level (L1)

name	description	source	language	No. lines
moldyn	molecule dynamics simulation	Chaos group	f77	660
mesh	structural simulation	Chaos group	C	932
magi	particle hydrodynamics	DoD	f90	9339
NAS-CG	sparse matrix-vector multiplication	NAS/NPB Serial v2.3	f77	1141

application	input size	source of input	exe. time
moldyn	256K particles, 27.4M interactions, 1 iteration	random initialization	53.2 sec
mesh	9.4K nodes, 60K edges, 20 iterations	provided by the Chaos group	8.14 sec
magi	28K particles, 253 cycles	provided by DoD	885 sec
NAS-CG	14K non-zero entries, 15 iterations	NASA/NPB Serial 2.3, Class A	48.3 sec

application	optimizations applied			program components measured
	locality grouping	regrouping	packing	
moldyn	+	V	V	subroutine $Compute\_Force()$
mesh	V	+	V	full application
magi	+	V	V	full application
NAS-CG	n/a	+	V/+	full application

Table 2: Applications, Input Sizes, and Transformations Applied

cache is 32KB in size and uses 32-byte cache lines and the second-level (L2) cache is 4MB with 128-byte cache lines. Both caches are two-way set associative. The R10K achieves 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.

The second table in Table 2 gives the input size for each application, the sources of the data inputs, and the execution time before applying optimizations. The working set is significantly larger than the L1 cache for all applications. $Mesh$, $Magi$ and $NAS-CG$ are a little bit larger than L2. $Moldyn$ has the largest data input and its data size is significantly greater than the size of L2.

We applied the three transformations in the following order: locality grouping, optimal data regrouping, dynamic data packing and packing optimizations. Since the access sequence is already transformed by locality grouping, we use consecutive packing for all cases because of the observation made in Section 2.2. (One test case, $NAS-CG$, accesses each element only once, therefore consecutive packing is optimal.) For each transformation applied, we measure its impact on execution time and the number of cache and TLB misses.

5.2 Transformations Applied

The third table in Table 2 lists, for each application, the optimizations applied and the program components measured. Each of the base programs came with one or more of the three optimizations done by hand. Such cases are labeled with a '+' sign in the table. The 'V' signs indicate the optimizations we added, except in the case of NAS-CG. The base program of $NAS-CG$ came with data packing already done by hand, but we removed it for the purpose of demonstrating the effect of packing. We do not consider hand-applied packing practical because of the complexity of transforming tens of arrays repeatedly at run-time for a large program.

Locality grouping and data regrouping were inserted by hand. Data packing of $moldyn$ and CG was performed automatically by our compiler given a one-line directive of packing. The same compiler packing algorithm was applied to $mesh$ by hand because our compiler infrastructure cannot yet compile C. Unlike other programs, $Magi$ is written in Fortran90 and computes the interaction list incrementally. We slightly modified the source to let it run through the Fortran77 front-end and inserted a loop to collect the overall data access sequence. Then our compiler successfully applied base packing transformation on the whole program. The application of the two compiler optimizations were semi-automatic: we inserted a 3-line loop to perform pointer update; and we annotated a few dependence-free loops which otherwise would not be recognized by the compiler due to the presence of procedural calls inside the them. All other transformations are performed by the compiler. The optimized packing reorganizes a total of 45 arrays in $magi$.

We refer to the original program as the base program and the transformed version with optimizations labeled 'V' as the optimized program. For NAS-CG, the base program refers to the version with no packing. Dynamic data packing is applied only once in each application except *magi* where data are repacked every 75 iterations.

5.3 Effect of Transformations

The four graphs of Figure 5 show the effect of the three transformations. The first plots the effect of optimizations on the execution speed. The first bar of each application is the normalized performance (normalized to 1) of the base version. The other bars show the performance after applying each transformation. Since not all transformations are necessary, an application may not have all three bars. The second bar, if shown, shows the speedup of locality grouping. The third and fourth bars show the speedup due to data regrouping and data packing. The other three graphs are organized in the same way, except that they are showing the reduction on the number of L1, L2 and TLB misses. The graphs include the miss rate of the base program, but the reduction is on the total number of misses, not on the miss rate.

Effect of Locality Grouping and Data Regrouping

Locality grouping eliminates over half of L1 and L2 misses in *mesh* and improves performance by 20%. In addition, locality grouping avails the program for data packing, which further reduces L1 misses by 35%. Without the locality grouping step, however, consecutive packing not only results in no improvement but also incurs 5% more L1 misses and 54% more L2 misses. This confirms the observation from our simulation study that locality grouping is critical for the later data optimization to be effective.

Data regrouping significantly improves *moldyn* and *magi*. *Magi* has multiple computation phases, optimal regrouping selectively groups 26 arrays into 6 arrays in order to achieve full cache utilization and maximal spatial reuse. As a result, the execution time is improved by a factor of 1.32 and cache misses are reduced by 38% for L1, 17% for L2, and 47% for TLB. By contrast, merging all 26 arrays improves performance by only 12%, reduces L1 misses by 35%, and as a side effect, increases L2 misses by 32%. Data regrouping is even more effective on *moldyn*, eliminating 70% of L1 and L2 misses and almost doubling the execution speed.

Effect of Dynamic Data Packing

Data packing is applied to all four applications after locality grouping and data regrouping. It further improves performance in all cases. For *moldyn*, packing improves performance by a factor of 1.6 and reduces L2 misses by 21% and TLB misses by 88% over the version after data regrouping. For $NAS-CG$, the speedup is 4.36 and the amount of reduction is 44% for L1, 85% for L2 and over 97% for TLB.

For *mesh* after locality grouping, packing slightly improves performance and reduces misses by additional 3% for L1 and 35% for L2. The main reason for the modest improvement on L1 is that the data granularity (24 bytes) is close to the size of L1 cache lines (32 bytes), leaving little room for additional spatial reuse. In addition, packing is directed by the traversal of edges, which does not work as well during the traversal of faces. The number of L1 misses is reduced by over 6% during edge traversals, but the reduction is less than 1% during face traversals. Since the input data set almost fits in L2, the significant reduction in L2 misses does not produce a visible effect on the execution time.

When applied after data regrouping on *magi*, packing speeds up the computation by another 70 seconds (12%) and reduces L1 misses by 33% and TLB misses by 55%. Because of the relatively small input data set, L2 and TLB misses are not a dominant factor in performance. As a result, the speed improvement is not as pronounced as the reduction in these misses.

Overall, packing achieves a significant reduction in the number of cache misses especially for L2 and TLB, where opportunities for spatial reuse are abundant. The reduction in L2 misses ranges from 21% to 84% for all four applications; the reduction in TLB misses ranges from 55% to 97% except for *mesh*, whose working set fits in TLB.

Packing Overhead and the Effect of Compiler Optimizations

The cost of dynamic data packing comes from the overhead of data reorganization and the cost of indirect memory accesses. The time spent in packing has a negligible effect on performance in all three applications we measured. Packing time is 13% of the time of one computation iteration in *moldyn*, and 5.4% in *mesh*. When packing is applied for every 20 iterations, the cost is less than 0.7% in *moldyn* and 0.3% in *mesh*. *Magi* packs data every 75 iterations and spends less than 0.15% of time on packing routines.

The cost of data indirection after packing can be mostly eliminated by two compiler optimizations described in Section 3.1. Figure 6 shows the effect of these two compiler optimizations on all four applications we tested.

The upper-left graph shows that, for *moldyn*, the indirections (that can be optimized away) account for 10% of memory loads, 22% of L1 misses, 19% of L2 misses and 37% of TLB misses. After the elimination of the indirections and the references to the run-time map, execution time was reduced by 27%, a speedup of 1.37. The improvement in *mesh* is even larger. In this case, the indirections account for 87% of the loads from memory, in part because *mesh* is written in C and the

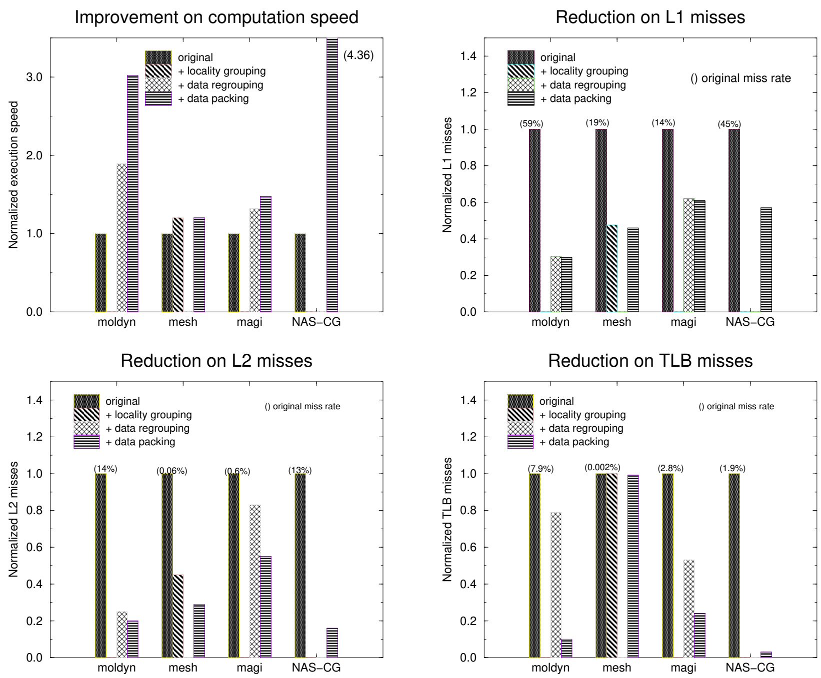

Figure 5: Effect of Transformations

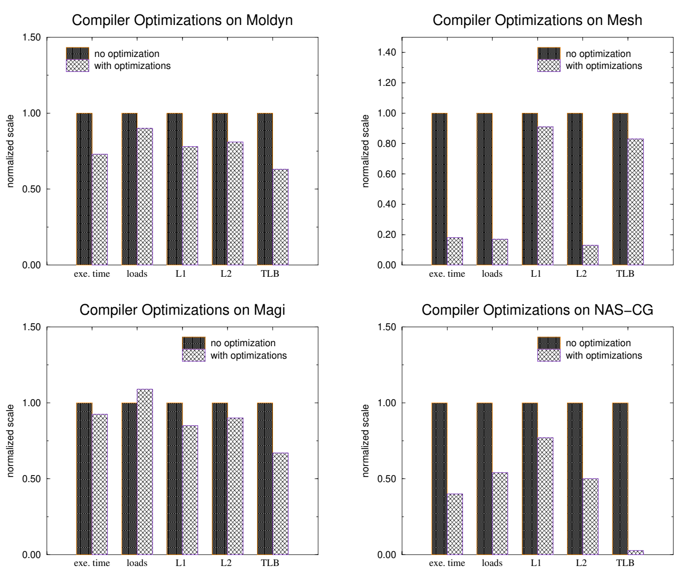

Figure 6: Effect of Compiler Optimizations

compiler does not do a good job of optimizing array references. Since the excessive number of memory loads dominates execution time, the compiler optimizations achieve a similar reduction (82%) in execution time. The number of loads is increased in *magi* after the optimizations because array alignment transforms 19 more arrays than the base packing, and not all indirections to these arrays can be eliminated. Despite the increased number of memory loads, the cache misses and TLB misses are reduced by 10% to 33%, and the overall speed is improved by 8%. For $NAS-CG$, the compiler recognizes that matrix entries are accessed in stride-one fashion and consequently, the compiler replaces the indirection accesses with direct stride-one iteration of the reorganized data array. The transformed matrix-vector multiply kernel has the equally efficient data access as the original hand-coded version. As a result, the number of loads and cache misses is reduced by 23% to 50%. The TLB working set fits in machine's TLB buffer after the optimizations, removing 97% of TLB misses. The execution time is reduced by 60%, a speedup of 2.47.

6 Related Work

To our knowledge, this work is the first study on the combination of run-time computation and data transformation to improve cache performance of irregular and dynamic applications. It is also the first to provide comprehensive compiler support for run-time data transformation.

Our work is close in spirit to the run-time parallelization work on dynamic applications. The Chaos group, led by Saltz[10], partitions computation and reorganizes data at run time in order to balance the computational load across processors and reduce communication in a parallel execution. Once computation is partitioned, the data accessed by each processor are grouped and placed in its local memory. However, the parallelization work did not include a general restructuring method to subsequently improve cache performance.

The Chaos group is also the first to use run-time computation transformation to improve cache performance. Das et al. used a reverse Cuthill Mcgee ordering to improve locality in a multi-grid computation[9]. Another method, domain partitioning, has been used to block computation for cache by Tomko and Abraham[21]. However, they found no overall improvement by blocking. Building on our work reported in this paper, Mellor-Crummey et al.[18] have employed space-fitting curve ordering to block dynamic computation. Domain partitioning and space-curve ordering are more powerful than locality grouping because they can block computation for a specific cache size, but they are also more expansive than locality grouping. In fact, locality grouping works without looking at domain data, i.e. coordinates of particles. For applications such as *mesh*, locality grouping is able to reduce miss rate to as low as 0.37%, leaving little room for further improvement with more expansive methods. An idea that is similar to locality grouping is used by Han and Tseng to improve parallel efficiency on a shared-memory machine[13]. They used owner-compute rule and assigned all updates of a particle to a single thread to avoid the cost of reduction among parallel processors.

The compiler support for dynamic data packing overcomes a serious limitation of all previous run-time methods, that is, their reliance on the knowledge of program structure and data domain. By comparison, our compiler exploits and optimizes data layout transformation without relying on domain knowledge of either the program or data. There has been recent work on hardware-based data reorganization by Carter at el.[6]. Their approach can be potentially more efficient because they use an additional processor to remap memory. However, compiler analysis similar to ours is necessary to effectively control such hardware features.

The goal of improving data reuse has been pursued for regular applications by loop and data transformations such as cache blocking[5, 22], memory order loop permutation[1, 12, 17], and data reshaping[8, 3, 15]. However, static loop and data transformations developed for regular applications cannot optimize dynamic computations where the data access pattern remains unknown until run time and changes during the computation.

Various static data placement schemes have been used to avoid cache conflicts and to improve spatial reuse. Thabit[20] studied data packing to reduce conflicts in cache. He used static program analysis to construct a proximity matrix and packed simultaneously used data into non-conflicting cache blocks. He proved that finding the optimal packing using a proximity matrix is NP-complete. Al-Furaih and Ranka modeled irregular data as graph nodes and used edges to link the data that are simultaneously accessed[2]. In addition, for programs with high-level data structures and dynamic memory allocation, profiling information has been used to analyze the order of access to both code and data. Seidl and Zorn[19] clustered frequently referenced objects, and Calder et al.[4] reordered objects based on their temporal relations.

Our work differs from such static data layout transformations in that we apply data packing at run time. Efficiently exploiting spatial reuse at run time is critical for applications in which data access order changes during the execution. For example, in a sparse-matrix code, the matrix may be iterated first by rows and then by columns. In scientific simulations, the computation order changes as the physical model evolves. In these cases, a fixed static data layout is not likely to perform well throughout the computation. Another difference is that our packing method does not explicitly use proximity or temporal relations of data. It is not yet established whether the data reordering methods based on proximity matrices or temporal relation graphs are cost effective at run time. The cost of constructing a complete proximity relationship can be prohibitively high, given a large number of data elements involved. The third difference is the granularity of the analysis and transformation. Profiling-based methods have the fixed granularity, i.e. the unit of memory allocation. The granularity of our analysis and transformation is individual array elements. In addition, our data transformation aligns and merges array elements that are attributes of the same particle.

7 Contributions

We have presented and evaluated three novel program and data transformations for improving the memory hierarchy performance of dynamic applications. The principal contribution of this paper is a demonstration of how compiler-generated computation and data reorganization at run time can improve temporal and spatial reuse. We examined two run-time transformations for this purpose.

- *Locality grouping* brings together all the interactions involving the same data element. It is inexpensive, yet very powerful, eliminating over 50% of cache misses in a full application. Furthermore, locality grouping is vital for the subsequent data transformation to be effective.

- *Dynamic data packing* improves spatial reuse by reorganizing the data structures so that data elements that are used together are close together in memory. Since optimal data packing is NP-complete, we have explored three different heuristic-based packing algorithms and found that simple consecutive packing performs extremely well when carried out after locality grouping. For all applications tested on SGI Origin2000, dynamic data packing reduced the number of L2 misses by 21% to 84% and the number of TLB misses by 55% to 97%. As a result, it improved overall program performance by a factor up to 4.36.

A second contribution of this paper is a compiler strategy for eliminating overhead associated with dynamic data packing. This support has been implemented in an interprocedural compiler at Rice. The compiler uses run-time data maps and data indirections to ensure the correctness of any run-time data transformation. It then employs two optimizations, pointer update and array alignment, to eliminate most of the data indirections after data reorganization. The compiler optimizations are very effective in removing packing overhead for all applications tested, improving performance by factors ranging from 1.08 to 5.56.

Acknowlegement

The implementation described here is based on the D System infrastructure at Rice University, a project led by John Mellor-Crummey and Vikram Adve. We depended especially on the scalar compiler version originated by Nat McIntosh. Dennis Moreau and William Eaton provided access to SGI Origin2000, and Ehtesham Hayder helped setting up application *magi*. We also wish to thank Keith Cooper and anonymous referees for helpful comments on the early drafts of this paper.

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] I. Al-Furaih and S. Ranka. Memory hierarchy management for iterative graph structures. In *Proceedings of IPPS*, 1998.

[3] J. Anderson, S. Amarasinghe, and M. Lam. Data and computation transformation for multiprocessors. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, CA, July 1995.

[4] B. Calder, K. Chandra, S. John, and T. Austin. Cache-conscious data placement. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, San Jose, Oct 1998.

[5] D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, White Plains, NY, June 1990.

[6] J. Carter, W. Hsieh, M. Swanson, L. Zhang, A. Davis, M. Parker, L. Schaelicke, L. Stoller, and T. Tateyama. Memory system support for irregular applications. In *Workshop on Languages, Compilers, and Runtime Systems for Scalable Computers*, May 1998.

[7] R. Chandra, D. Chen, R. Cox, D.E. Maydan, and N. Nedeljkovic. Data distribution support on distributed shared memory multiprocessors. In *Proceedings of '97 Conference on Programming Language Design and Implementation*, 1997.

[8] M. Cierniak and W. Li. Unifying data and control transformations for distributed share d-memory machines. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, La Jolla, 1995.

[9] R. Das, D. Mavriplis, J. Saltz, S. Gupta, and R. Ponnusamy. The design and implementation of a parallel unstructured euler solver using software primitives. In *Proceedings of the 30th Aerospace Science Meeting*, Reno, Navada, January 1992.

[10] R. Das, M. Uysal, J. Saltz, and Y.-S. Hwang. Communication optimizations for irregular scientific computations on distributed memory architectures. *Journal of Parallel and Distributed Computing*, 22(3):462–479, September 1994.

[11] C. Ding. Improving effective bandwidth on machines with complex memory hierarchy. Thesis Proposal, Rice University, November 1998.

[12] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformations. In *Proceedings of the First International Conference on Supercomputing*. Springer-Verlag, Athens, Greece, June 1987.

[13] H. Han and C.-W. Tseng. Improving compiler and run-time support for adaptive irregular codes. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, October 1998.

[14] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[15] T. E. Jeremiassen and S. J. Eggers. Reducing false sharing on shared memory multiprocessors through compile time data transformations. pages 179–188, July 1995.

[16] D. G. Kirkpatrick and P. Hell. On the completeness of a generalized matching problem. In *The Tenth Annual ACM Symposium on Theory of Computing*, 1978.

[17] 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.

[18] J. Mellor-Crummey, D. Whalley, and K. Kennedy. Improving memory hierarchy performance for irregular applications. Technical Report TR 99-336, Department of Computer Science, Rice University, Feburary 1999.

[19] M. L. Seidl and B. G. Zorn. Segregating heap objects by reference behavior and lifetime. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, San Jose, Oct 1998.

[20] K. O. Thabit. *Cache Management by the Compiler*. PhD thesis, Dept. of Computer Science, Rice University, 1981.

[21] K. A. Tomko and S. G. Abraham. Data and program restructuring of irregular applications for cache-coherent multiprocessors. In *Proceedings of '94 International Conference on Supercomputing*, 1994.

[22] M. E. Wolf and M. Lam. A data locality optimizing algorithm. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, Toronto, Canada, June 1991.