Improving Memory Hierarchy Performance for Irregular Applications

John Mellor-Crummey†, David Whalley‡, Ken Kennedy†

† Department of Computer Science, MS 132
Rice University
6100 Main
Houston, TX 77005
e-mail: {johnmc,ken}@cs.rice.edu
phone: (713) 285-5179

‡ Computer Science Department
Florida State University
Tallahassee, FL 32306-4530
e-mail: whalley@cs.fsu.edu
phone: (850) 644-3506

Abstract

The gap between CPU speed and memory speed in modern computer systems is widening as new generations of hardware are introduced. Loop blocking and prefetching transformations help bridge this gap for regular applications; however, these techniques don't deal well with irregular applications. This paper investigates using data and computation reordering strategies to improve memory hierarchy utilization for irregular applications on systems with multi-level memory hierarchies. We introduce multi-level blocking as a new computation reordering strategy and present novel integrations of computation and data reordering using space-filling curves. In experiments that applied a combination of data and computation reorderings to two irregular programs, overall execution time dropped by about a factor of two.

1. Introduction

The gap between CPU speed and memory speed is increasing rapidly as new generations of computer systems are introduced. Multi-level memory hierarchies are the standard architectural design used to bridge this memory access bottleneck. As the gap between CPU speed and memory speed widens, systems are being constructed with deeper hierarchies. Achieving high performance on such systems requires tailoring the reference behavior of applications to better match the characteristics of a machine's memory hierarchy. Techniques such as loop blocking [1, 2, 3, 4, 3, 5] and data prefetching [4, 6, 7] have significantly improved memory hierarchy utilization for regular applications. A limitation of these techniques is that they don't deal well with irregular applications. Improving performance for irregular applications is extremely important since they often access large amounts of data and can be quite time consuming. Furthermore, such applications are very common in science and engineering.

Irregular applications are characterized by patterns of data and computation that are unknown until run time. In such applications, accesses to data often have poor spatial and temporal locality, which leads to ineffective use of a memory hierarchy. Improving memory hierarchy performance for irregular applications requires addressing problems of both latency and bandwidth. Latency is a problem because poor temporal and spatial reuse result in elevated cache and translation lookaside buffer (TLB) miss rates. Bandwidth is a problem because indirect references found in irregular applications tend to have poor spatial locality. Thus, when accesses cause blocks of data to be fetched into various levels of the memory hierarchy, items within a block are either referenced only a few times or not at all before the block is evicted due to conflict and/or capacity misses, even though these items will be referenced later in the execution.

One strategy for improving the memory hierarchy performance for such applications is to reorder data dynamically at the beginning of a major computation phase. This approach assumes that the benefits of increased locality through reordering will outweigh the cost of the data movement. Data reordering can be particularly effective when used in conjunction with a compatible computation reordering. The aim of data and computation reordering is to more effectively utilize bandwidth at different levels of the memory hierarchy by (1) increasing the probability that items in the same block will be referenced close together in time and (2) increasing the probability that items in a block will be reused more extensively before the block is replaced. This paper concentrates on investigating the effectiveness of such strategies in the presence of multi-level memory hierarchies. We introduce multi-level blocking as a new computation reordering strategy for irregular applications and consider novel integrations of computation reordering with data reordering based on space-filling curves.

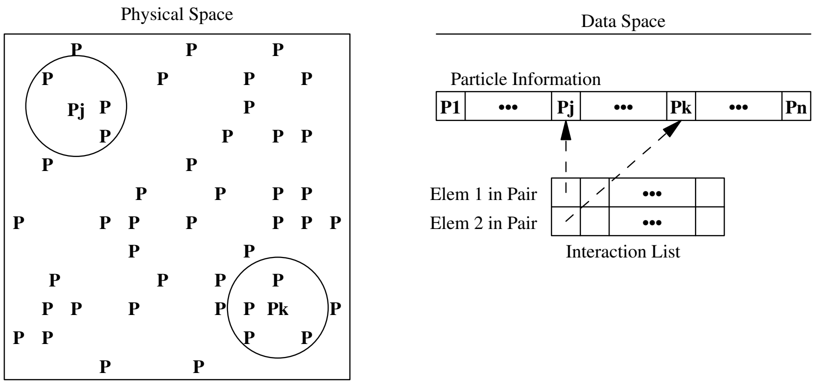

Figure 1: A Classical Irregularly Structured and Adaptive Application

A common class of irregular applications considers particles or mesh elements in spatial neighborhoods. Figure 1 shows a sample n-body simulation that we use as an example throughout the paper. Although we explain our techniques in terms of this example, they apply more broadly to any irregular application simulating physical systems in two or more dimensions. Our sample n-body simulation considers particles within a defined volume, represented here as a two dimensional area for simplicity. Each particle interacts with other particles within a specified cutoff radius. Particles $\mathbf{P_j}$ and $\mathbf{P_k}$ are shown in the physical space along with a cutoff radius surrounding each particle. Interactions are between a particle and other particles within its cutoff radius. The particles can change positions in the physical space of the problem. To adapt to these changes, the application requires periodic recalculation of which particles can interact.

Figure 1 also shows the problem data space for this sample application. The information for each particle includes its coordinates in the physical space and other attributes, such as velocity and the force exerted upon it. The interaction list indicates the pairs of particles that can interact. The data for the particles is irregularly accessed since the order of access is determined by the interaction list. The number of interactions is typically much greater than the number of particles. Note that there are many possible variations on how the program data space can be organized.

The remainder of this paper has the following organization. First, we introduce related work that uses blocking, data reordering, and space-filling curves to improve the performance of computer systems. Second, we outline the general data and computation reordering techniques that we consider in this paper. Third, we describe two irregular programs, explain how we apply specific combinations of data and computation reordering techniques, and present the results of applying these reordering techniques on these programs. Fourth, we discuss future work for applying data and computation reorderings to irregular applications. Finally, we present a summary and conclusions of the paper.

2. Related Work

Blocking for improving the performance of memory hierarchies has been a subject of research for several decades. Early papers focused on blocking to improve paging performance [8, 9], but recent work has focused more narrowly on improving cache performance [2, 3, 4, 5]. Techniques similar to blocking have also been effectively applied to improvement of reuse in registers [1]. Most of these methods deal with one level of the memory hierarchy only, although the cache and register techniques can be effectively composed. A recent paper by Navarro *et al.* examines the effectiveness of multi-level blocking techniques on dense linear algebra [11] and a paper by Kodukula *et al.* presents a data-centric blocking algorithm that can be effectively applied to multilevel hierarchies [12].

The principal strategy for improving bandwidth utilization for regular problems, aside from blocking for reuse, has been to transform the program to increase spatial locality. Loop interchange is a standard approach to achieving stride-1 access in regular computations. This transformation has been specifically studied in the context of memory hierarchy improvement by a number of researchers [10, 13, 14].

As described earlier, data reordering can be used to reduce bandwidth requirements of irregular applications. Ding and Kennedy [15] explored compiler and run-time support for a class of run-time data reordering techniques. They examine an access sequence and use it to reorder data to increase the spatial locality as the access sequence is traversed. They consider only a very limited form of computation reordering in their work. Namely, for computations expressed in terms of an access sequence composed of tuples of particles or objects, they apply a grouping transformation to order tuples in the sequence to consider all interactions involving one object before moving to the next. This work did not specifically consider multilevel memory hierarchies although it did propose a strategy for grouping information about data elements to increase spatial locality, which has the side effect of improving TLB performance. In our work, we applied this grouping strategy before taking baseline performance measurements. Also, we evaluate Ding and Kennedy's best strategy, first-touch reordering, along with other strategies.

In recent years, space-filling curves have been used for managing locality for both regular and irregular applications. Space-filling curves are continuous, nonsmooth curves that pass arbitrarily near every point in a finite space of arbitrary dimension. First constructed by Peano in 1890, space-filling curves can be used to map between a 1-dimensional coordinate space and any d-dimensional coordinate space, where $d \geq 2$. Such curves establish a one-to-one correspondence between points in the different coordinate spaces. Mapping the coordinates of a point in d-dimensional space to its 1-dimensional counterpart is accomplished through a sequence of bit-level logical operations on the coordinates in each dimension. A Hilbert space-filling curve is one such mapping. Figure 2 shows a fifth-order Hilbert curve in two dimensions. An important property of this curve, is that its recursive structure preserves locality: points close in the original d-dimensional space are typically close along the curve. In particular, the successor of any point along the curve is one of its adjacent neighbors along one of the coordinate dimensions.[1]

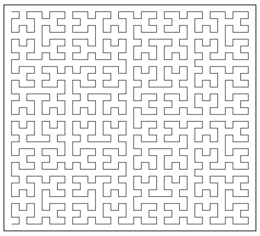

Figure 2: Fifth-order Hilbert curve through 2 dimensions.

Space-filling curves have been used to partition data and computation among processors in a parallel computer system. They have been applied in problems domains including molecular dynamics [18], graph partitioning [19], and adaptive mesh refinement [20]. Ordering the data elements by their position along a space-filling curve and assigning each processor a contiguous range of elements of equal (possibly weighted) size is a fast partitioning technique that tends to preserve physical locality in the problem domain. Namely, data elements close together in physical space tend to be in the same partition. Ou *et al.* [19] present results that show that other methods such as recursive spectral bisection and reordering based on eigenvectors can produce partitionings with better locality according to some metrics; however, the differences among the methods (in terms of the locality of partitionings produced) diminished when these methods were applied to larger problem sizes. Also, they found that using space-filling curves was orders of magnitude faster than the other methods they studied.

Thottethodi *et al.* [21] explored using space-filling curves to improve memory hierarchy performance for dense matrix multiplication. They ordered matrix elements according to a 2D space-filling curve rather than the usual row-major or column-major order to improve the cache performance of Strassen's matrix multiplication algorithm. They found the hierarchical locality resulting from the space-filling curve order to be a good match for the recursive structure of Strassen's algorithm.

[1] For more details about the history of space-filling curves, the types of curves, their construction, and their properties, see Sagan [16] and Samet [17].

Al-Furaih and Ranka [22] explored several strategies for data reordering to improve the memory hierarchy performance of iterative algorithms on graphs. They evaluated several data reordering methods including graph partitioning, space-filling curves, and breadth-first traversal orders. They measured improvements in execution time of 20-50% of the computational kernels from their data reordering strategies. Our work differs from theirs principally in that we consider approaches that combine data and computation reordering, whereas they consider data reordering exclusively.

3. Data Reordering Approaches

A data reordering approach involves changing the location of the elements of the data, but not the order in which these elements are referenced. Consider again the data space shown in Figure 1. Data reordering would involve changing the location of the elements within the particle information and updating the interaction list to point to the new particle locations. By placing data elements near one another that are referenced close in time, data reordering approaches can improve spatial locality. Temporal locality would not be affected since the order in which data elements are accessed remains unchanged. The following subsections describe the data reordering approaches investigated in this work.

3.1. First Touch Data Reordering

First-touch data reordering is a greedy approach for improving spatial locality of irregular references. Consider Figure 3, which represents the data space in Figure 1 before and after data reordering using the first-touch approach. A linear scan of the interaction list is performed to determine the order in which the particles are first touched. The particle information is reordered and the indices in the interaction list now point to the new positions of the particles. However, the order in which the particles are referenced is unchanged. The idea is that if two particles are referenced near each other in time in the interaction list, then they should be placed near each other in the particle list. The advantage of first-touch data reordering is that the approach is simple and efficient since it can be accomplished in linear time. A disadvantage is that the computation order (interaction list in Figure 3) must be known before reordering can be performed.

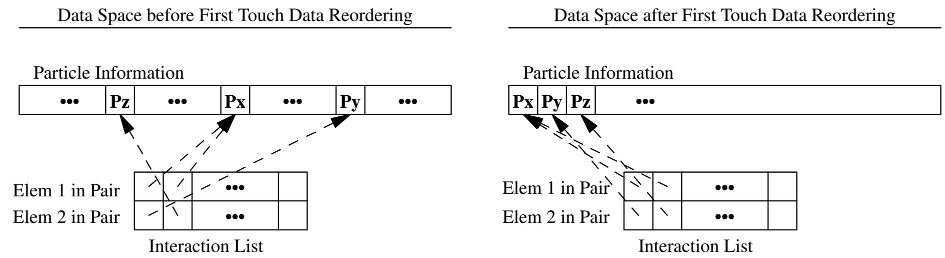

Figure 3: Data Reordering Using a First Touch Approach

3.2. Space Filling Curve Data Reordering

Figure 4 shows an example data space before and after data reordering using a space-filling curve. Assume that the first three particles on the curve are $\mathbf{P_x}$, $\mathbf{P_y}$, and $\mathbf{P_z}$. To use a k-level space-filling curve to reorder data for particles whose coordinates are represented with real numbers, several steps are necessary. First, each particle coordinate must be normalized into a k-bit integer. The integer coordinates of each particle's position are converted into a position along the space-filling curve by a sequence of bit-level logical operations. The particles are then sorted in ascending order by their position along the curve. Sorting particles in space-filling curve order tends to increase spatial locality. Namely, if two particles are close in physical space, then they tend to be close on the curve. One advantage of using a space-filling curve for data reordering is that data can be reordered prior to knowing the order of the computation. This allows some computation reorderings to be accomplished with no overhead. For instance, if the data is reordered prior to establishing the order of the references (e.g. an interaction list), then the reference order will be affected if it is established as a function of the order of the data. A potential disadvantage of using space-filling curves is that it is possible that the reordering may require more overhead than a first-touch reordering due the sort of the particle information. Of course, the relative overheads of the two approaches would depend on the number of data elements versus the number of references to the data.

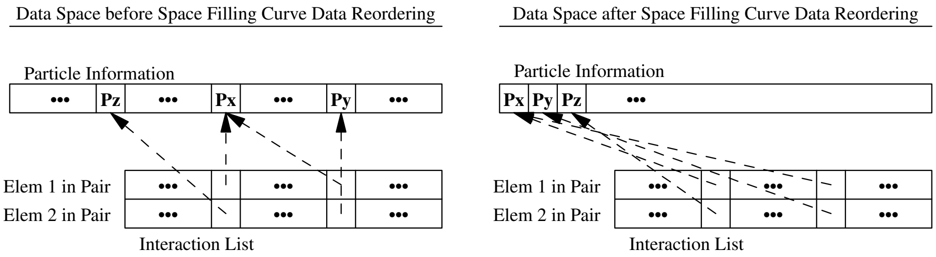

Figure 4: Data Reordering Using a Space Filling Curve Approach

4. Computation Reordering Approaches

A computation reordering approach involves changing the order in which the data elements are referenced, but not the locations in which these data elements are stored. Consider again the data space shown in Figure 1. Computation reordering would involve reordering the pairs of elements within the interaction list. The particle information from which values are accessed for the computation would remain unchanged. Computation reordering approaches can improve both temporal and spatial locality by reordering the accesses so that the same or neighboring data elements are referenced close in time. The following subsections describe the computation reordering approaches considered in this work.

4.1. Space Filling Curve Computation Reordering

Reordering the computation in space-filling curve order requires determining the position along the curve for each data element and using these positions as the basis for reordering accesses to the data elements. Figure 5 shows an example data space before and after computation reordering. Assume that the first three particles in space-filling curve order are $\mathbf{P_x}$, $\mathbf{P_y}$, and $\mathbf{P_z}$. To reorder the computation, entries in the interaction list, as shown in Figure 5, are sorted according to the space-filling curve position of the particles they reference. The order of the particle information itself remains unchanged. Combined with a space-filling curve based data reordering, a space-filling curve based computation reordering can lead to improved temporal locality. For instance, if particle X interacts with particle Y, then it is likely that particle Y will be referenced again soon since Y in turn will interact with other particles. Without reordering the data in space-filling curve order, it is unclear what benefits will be achieved by space-filling curve based computation reordering alone.

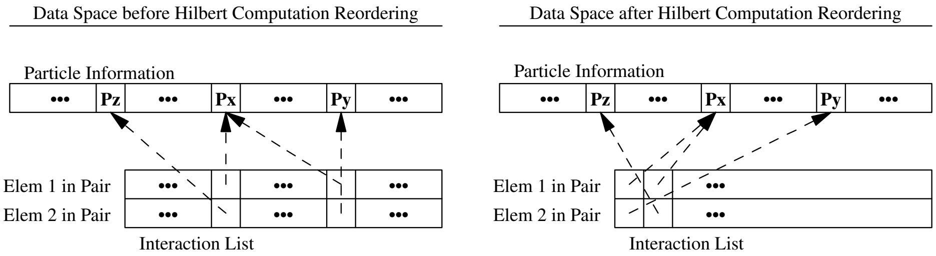

Figure 5: Example of Computation Reordering

4.2. Computation Reordering by Blocking

As described earlier in the paper, blocking computation via loop nest restructuring has been used successfully to improve memory hierarchy utilization by regular applications for multi-level memory hierarchies. Here we describe how blocking can be used as a computation reordering technique for some irregular applications as well.

In terms of our n-body example, the following loop nest is an abstract representation of the natural computation ordering for the given data order:

```
FOR i = 1 to number of particles DO
    FOR j in the set particles_that_interact_with[i] DO
        process interaction between particles i and j
```

Blocking first assigns a block number to each particle based on its memory location in the vector of particles. Then, rather than considering all interactions for each particle at once, we consider all interactions between particles in each pair of blocks, as block pairs are traversed in natural order. This is depicted in the following code fragment.

```
FOR i = 1 to number of blocks of particles DO
    FOR j in the set blocks_that_interact_with[i] DO
        process interactions between all interacting
        particle pairs with one particle in block i and
        one in block j
```

To extend this strategy to multiple levels of the memory hierarchy, we choose a blocking factor for each level. Just as in blocking approaches for regular applications, the size of an appropriate blocking factor depends on the size of the respective cache at that level of the memory hierarchy, its associativity, and the amount of other data that is being referenced in the computation. For instance, the interaction list will be accessed in the n-body computation outlined in Figure 1 while the particle information is being referenced. This would affect the blocking factors.

To implement this reordering in irregular applications where the reference order is explicitly specified by an interaction list, one can simply sort the interactions into the desired order based on the block numbers of the data elements they reference. For one level, one could simply sort the interaction pairs by the block number of the second particle in each pair, then sort by the block number of the first particle in each pair. This corresponds to a lexicographic sort of the block numbers for each pair [23].

To extend this strategy to multiple levels requires three steps. First, form an integer vector of block numbers for each particle, one block number for each level of the memory hierarchy. Second, interleave the vectors for each of the particles in an interaction pair. Third, apply a lexicographic sort to achieve the final ordering of the interaction list. This has the effect of first sorting by the smallest block size, corresponding to L1 cache, followed by sorts of the block numbers for each of the levels of the memory hierarchy in order of increasing size (e.g. L1, TLB, L2). In section 5.1, we explain how we achieve the effect of this sort rapidly in practice.

5. Applying the Techniques

The following subsections describe our experiences in applying data and computation reordering techniques to improve the performance of the *moldyn* and *magi* programs. *Moldyn* is a synthetic benchmark and *magi* is a production code. These programs are described in more detail in the following subsections. Both are irregular programs that exhibit poor spatial and temporal locality, which are typical problems exhibited by this class of applications.

We chose to perform our experiments on an SGI O2 workstation based on the R10000 MIPS processor since it provides hardware counters that enable collection of detailed performance measurements. Both programs were compiled with the highest level of optimization available for the native C and Fortran compilers.[2] Table 1 displays the configurations of the different levels of the memory hierarchy on this machine. Each entry in the TLB contains two virtual to physical page number translations, where each page contains 4KB of data. Thus, the 8KB block size for the TLB is the amount of addressable memory in two pages associated with a TLB entry.

5.1. The *Moldyn* Benchmark

Moldyn is a synthetic benchmark for molecular dynamics simulation. The computational structure in *moldyn* is similar to the nonbonded force calculation in CHARMM [24], and closely resembles the structure represented in Figure 1 of the paper. An iteraction list is constructed for all pairs of interactions that are within a specified cutoff radius. These interactions are processed every timestep and are periodically updated due to particles changing their spatial location.

[2] Although these compilers can insert data prefetch instructions to help reduce latency, prefetching is less effective for irregular accesses because prefetches are issued on every reference rather than every cache line [7]. Our experience was that data prefetching support in the SGI Origin C and Fortran compilers did not improve performance for the applications we studied and we did not use it in our experiments.

Cache Type	Cache Configuration		
	Size of Cache	Associativity	Block Size
Primary Data Cache	32KB	2-way	32B
Secondary Data Cache	1MB	2-way	128B
Translation Lookaside Buffer	512KB	64-way	8KB

Table 1: SGI O2 Workstation Cache Configurations

A high-level description of the computation for *moldyn* is shown in Figure 6. The time-consuming portion of the algorithm is the inner **FOR** loop in Figure 6 and is encapsulated in the *computeforces* function in the benchmark. This function traverses the interaction list performing a force calculation for each pair of particles. We applied different data and computation restructuring techniques in an attempt to make the *computeforces* function more efficient.

```
Randomly initialize the coordinates of each of the particles.
FOR N time steps DO
    Update the coordinates of each particle based on their force and velocity.
    Build an interaction list of particles that are within a specified radius every 20th time step.
    FOR each pair of particles in the interaction list DO
        Update the force on each of the particles in the pair.
    Update the velocities of each of the particles.
Print the final results.
```

Figure 6: Structure of the Computation in *Moldyn*

For our experiments, we set the number of particles to 256,000, which resulted in over 27 million interactions. We chose this large problem size to cause the data structures to be larger than the secondary cache and the amount of memory that can be contained in the pages associated with the TLB. Figure 7 depicts the data structures used in the *computeforces* function. The coordinates and forces have three elements for each particle since the physical space of the problem is in three dimensions. The length of the interaction list was long enough to contain all interacting pairs of particles. Each of the elements of the coordinates and forces are double precision values and the interaction list elements are integers used as indices into the coordinate and force arrays.

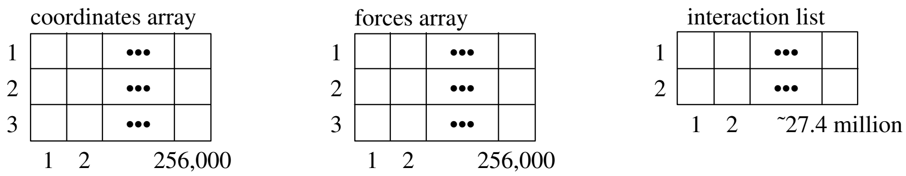

Figure 7: Main Data Structures in the *Moldyn* Benchmark

To make the *moldyn* benchmark more amenable to performing experiments with a large number of particles, we changed the approach for building the interaction list. Previously, a straightforward algorithm with $O(n^2)$ complexity was used to find all the interacting pairs of particles that were within the specified cutoff radius. We used an approach of dividing the physical space into cubes, where the length of each cube side was the size of the cutoff radius. We then assigned each particle to its respective cube. For a given particle, only the particles in current and immediate surrounding cubes had to be checked. (This is a well-known technique that is used by the *magi* application as well.) This allowed the interaction list to be built in a couple of minutes instead of several hours.

Before performing experiments with data and computation reorderings, we applied two transformations to remove orthogonal memory hierarchy performance problems.

(1) We interchanged the dimensions of the coordinates and the forces arrays so information for each particle would be contiguous in memory.

(2) We fused the coordinates and forces together (approximating an array of structures) to provide better spatial locality.

The purpose of this static program restructuring was to establish an aggressive performance baseline for our experiments. In our results below, all of our performance comparisons are with respect to this tuned "Original" program.

Table 2 shows information about misses in the caches and the TLB for the original *moldyn* benchmark and after performing both data and computation reordering using a Hilbert space-filling curve. Even after applying the data and computation reorderings based on the Hilbert curve, the high TLB miss ratio indicates poor utilization of the TLB. To investigate the nature of the poor memory hierarchy performance, we used a simulator to collect a TLB miss trace for the application. Figure 8 shows a plot of 10,000 TLB misses when a Hilbert curve was used to reorder the particle data and interaction computations. In the plot the block numbers are the tags of the addresses used to access the TLB and the interaction numbers indicate on which interaction each miss occurred. In the figure the dense band containing most of TLB misses plotted is over 100 blocks wide. This band slowly rises as the interaction numbers increase. The fact that most misses are in a narrow bands shows the effectiveness of reordering the data and computation in a Hilbert space-filling curve order. However, the number of entries in this TLB is 64. Thus, the entire band cannot fit in the TLB. This plot led us to realize that multi-level blocking would be useful for improving memory hierarchy utilization.

To accomplish multi-level blocking, the interaction list must be reordered to match the characteristics of the memory hierarchy of the target machine (as described in Section 4.2). Even though lexicographic sort as described in section 4.2 can be implemented in linear time, the interaction list must be reordered twice for each level of the memory hierarchy, which could be quite time consuming. To address this problem, we implemented an approach where the desired ordering of the interaction list could typically be attained in fewer passes, at the expense of space.

Figure 9 depicts how we achieve this effect. We partition the physical space for the particle information into blocks that are of the size of the blocking factor for the smallest level of the memory hierachy. For this blocking factor each interaction pair has two block numbers that uniquely identify the combination of blocks for the elements in the pair. These block numbers can be used as indices into a matrix of bins, where each bin contains a list of interaction pairs representing the combination of blocks associated with the elements in the pair. For instance, an interaction pair in which the first element in the pair has block number i and the second element has block number j would be placed in the interaction list associated with $bin[i][j]$. If the interaction pairs have been constructed so that the first element always has a lower particle number than the second element, then only the lower right half of the matrix of bins is needed. We fill these bins in two linear passes. The first pass determines the number of pairs associated with each bin and allocates space for each bin's list of interactions. The second pass copies the interaction pairs into the appropriate bin.

Once the interactions have been placed in their appropriate bins, we form a new interaction list by traversing the bins in the appropriate blocked order. Figure 10 depicts a multi-level blocking order for an interaction list,

Cache Type	Original Misses	Original Miss Ratio	Hilbert Miss Ratio
L1 Cache	1,605,701,730	0.16825	0.15747
L2 Cache	993,370,676	0.07039	0.03786
TLB	671,724,086	0.61865	0.36834

Table 2: Miss Information

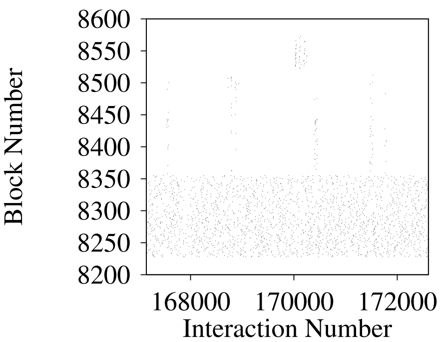

Figure 8: A Strip of 10,000 TLB Misses after Reordering the Data and the Computation in Hilbert Order

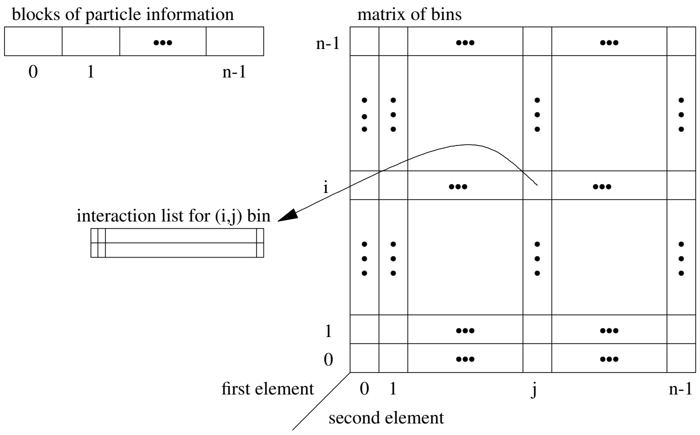

Figure 9: Data Structure Representing Combinations of Blocks for Interaction Pairs

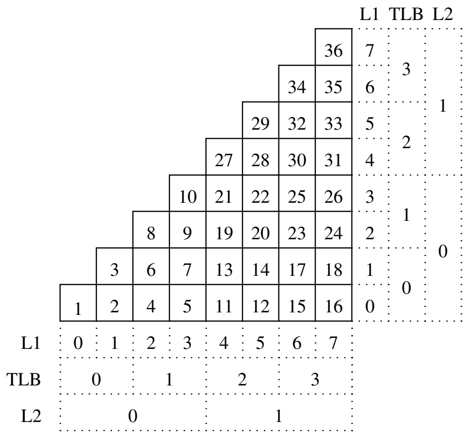

Figure 10: Order of Block Traversal to Create a New Interaction List

where only the lower right half of the matrix of bins is referenced. The numbers outside the bins indicate the block numbers for the three levels of the memory hierarchy. The data in this example spans eight L1 blocks. The numbers inside the bins indicate the order in which the bins are traversed. To keep the figure small, the blocking factor shown for the L2 cache is twice the size of the blocking factor for the TLB, which in turn is twice the size of the L1 blocking factor. We found through experimentation that the best blocking factors for *moldyn* were actually one half the cache size for the L1 and L2 caches and one quarter the TLB size for the TLB. All interactions within an L1 block are processed before proceeding onto the next L1 block, all L1 blocks within a TLB block are processed before the next TLB block, and all TLB blocks within an L2 block are processed before the next L2 block. We performed the blocked traversal of the bins using a set of nested loops that is similar to blocked loops for regular applications. Compared with using a single call to quicksort with a multi-level comparison function to perform the multi-level sort of the particle interaction pairs, our approach using of blocking the interaction list as described here was faster by a factor of 23.

Table 3 shows the results for applying the different combinations of data and computation reorderings to *moldyn* on an SGI O2 workstation. These results show ratios of end-to-end performance as compared to the original execution of *moldyn* without any data or computation reordering. The omission of the combination of using data reordering by first touch with computation reordering by blocking is intentional. First-touch data reordering requires knowing the order in which the data is referenced. Blocking requires knowing the addresses of each of the data elements. If data reordering was applied first, then blocking would change the order of the references and the benefits from the first-touch ordering would be affected. Likewise, if computation reordering by blocking were applied first, then first-touch reordering would affect the addresses of the data elements and ruin the effect from blocking.

There are several aspects of the results that are worth noting. First, the impact of data and computation reordering generally had a more significant effect on caches with a larger block or line size. Thus, the impact on the TLB misses was most significant and was least significant on primary cache misses. In fact, the primary cache block size was only 32 bytes and the size of the information for a single particle was 48 bytes, which limited the benefits of increased spatial locality. Second, a combination of data and compution reorderings typically performed better than using a specific type of data or computation reordering in isolation. The combination of data reordering using Hilbert space-filling curves to improve spatial locality and computation reordering using blocking to improve temporal locality was the most effective. This strategy reduced the miss ratios for L1 cache from 16.8% to 13.0%, for L2 cache from 7.0% to 2.6%, and for TLB from 61.9% to 8.1%.[3] In terms of reducing execution cycles, this method

[3] It is worth noting that since we are measuring end-to-end performance, the miss rates quoted for executions with reordering include all misses incurred performing the reordering as well as misses during computation. When we consider only the performance of the *computeforces* routine, we measured improvements of a factor of 20 for L2 misses and a factor of over 1000 in TLB misses.

Data Reordering	Computation Reordering	L1 Cache Misses	L2 Cache Misses	TLB Misses	Cycles
First Touch	None	0.87819	0.75985	0.31204	0.79366
Hilbert	None	1.02611	1.03930	0.85230	1.03114
None	Hilbert	1.03891	0.70817	0.99184	0.85828
None	Blocking	1.24393	0.40702	0.20846	0.57021
First Touch	Hilbert	0.98358	0.50547	0.23578	0.71565
Hilbert	Hilbert	0.93594	0.53843	0.59510	0.66968
Hilbert	Blocked	0.77293	0.27773	0.13129	0.45767

Table 3: Results of the Different Data and Computation Reorderings for *Moldyn*
(Ratios as Compared to the Original Measurements)

was 12% more effective than any of the other strategies, reducing overall cycles by over a factor of 2. It is worth noting that the second best result we obtained used computation blocking alone. Finally, it appears that the reduction of cycles was most closely correlated with the reduction in secondary cache misses. While there were fewer total misses in the secondary cache as compared to the primary cache, we suspect that the longer miss penalty in the secondary cache was less likely to be overlapped with other useful work and thus more likely to stall the processor.

5.2. The *Magi* Application

The *magi* application is a particle code used by the U.S. Air Force for performing hydrodynamic computations that focus on interactions of particles in spatial neighborhoods. The computational domain consists of objects comprised of particles and void space. A 3D rectangular space containing particles is divided into boxes, where the neighboring particles within a sphere of influence of a given particle are guaranteed to be in the same box or an adjacent box. For our experiments, we used DoD-provided test data involving 28,000 particles. For this test case, the size of the data structures is larger than the secondary cache and the amount of memory that can be contained in the pages associated with the TLB. A high-level description of the computation for *magi* is given in Figure 11.

```
Read in the coordinates of each of the particles.
FOR N time steps DO
    FOR each particle i DO
        Create an interaction list for particle i containing neighbors within the sphere of influence.
        FOR each particle j within this interaction list DO
            Update information for particle j.
Print the final results.
```

Figure 11: Structure of the Computation in *Magi*

Just as in the *moldyn* benchmark, we tuned the *magi* application to improve memory hierarchy performance to provide a more aggressive baseline for our experiments.

(1) We transposed several arrays associated with particle information so this information would be contiguous in memory.

(2) We fused some arrays together (approximating an array of structures) to provide better spatial locality when different kinds of particle information are referenced together.

Unlike the *moldyn* benchmark, a separate interaction list is created for each particle on each time step and is discarded after being used once. There is never an explicit representation of all the interactions. Therefore, computation reordering techniques that require reordering of the interaction list as presented in the *moldyn* benchmark would not be applicable for *magi*. Likewise, some types of data reordering cannot be accomplished in the same manner since there is no persistent representation of an interaction list that can be updated to point to the new location of the particles. Therefore, we used the following approaches to accomplish data and computation reordering for *magi*.

(1) We used an indirection vector containing the new positions of the particles when applying data reordering without computation reordering so the order in which the particles were referenced would be unaffected. This requires an additional level of indirection each time information about a particle is referenced, which can potentially have an adverse effect on both the performance of the memory hierarchy and the execution cycles.

(2) Data reordering using a space-filling curve does not depend on the order of the interactions and was performed before the first time step. First-touch data reordering was accomplished by (a) collecting the order of the references during the first time step across the different particle interaction lists and (b) reordering the particles before they are referenced on the second time step.

(3) When applying computation reordering, we simply did not use the indirection vector. Thus, the order of a subsequently generated interaction list is affected by the data reordering of the particle information.

(4) We composed a data reordering using a Hilbert space-filling curve followed by a data reordering using a first-touch approach without using an indirection vector to cause computation reordering. Placing the particles in Hilbert order results in a space-filling curve based computation order, which increases the likelihood that consecutive particles being processed will have many common neighbors in their interaction lists and improves temporal locality. Applying a first-touch reordering to the space-filling curve based computation order after the first time step greedily increases spatial locality. Note this approach is similar to applying computation reordering using a Hilbert space-filling curve approach and data reordering using a first-touch approach as was accomplished in *moldyn*. The only difference is that the interaction lists in *magi* were established at the beginning of each time step, which results in the first-touch data reordering also affecting the computation order.

Table 4 shows the results of applying the combinations of data and computation reorderings that were beneficial for the *magi* application. Several of the combinations of data and computation reorderings applied to the *moldyn* benchmark are not shown in this table for a couple of reasons. First, we found that applying data reordering only for *magi* degraded performance. The cost of accessing data through an indirection vector outweighed the benefits that were achieved by reordering data. One should note that data reordering without computation reordering can achieve benefits as shown for *moldyn* in Table 3. However, achieving such benefits may require that there is an inexpensive method to access the reordered data, such as updating an interaction list once to refer to the new data locations rather than incurring the cost of dereferencing an element of the indirection vector on each data reference. Second, the combinations of data and computation reorderings were also restricted by the fact that the interaction list for a particle was regenerated on each time step. Regeneration of the interaction lists prevented direct computation reordering. Likewise, separate and small interaction lists for each particle made the use of blocking inappropriate.

There are some interesting aspects of the results in Table 4. First, the decrease in execution cycles seems most closely correlated with the reduction of L2 cache misses, as was the case with *moldyn*. Second, the composition of Hilbert space-filling curve reordering followed by a first-touch reordering resulted in the best performance of the different combinations, reducing overall execution time by nearly a factor of 2.

Data Reordering	Computation Reordering	L1 Cache Misses	L2 Cache Misses	TLB Misses	Cycles
First Touch	First Touch	0.45200	0.27677	0.51212	0.57275
Hilbert	Hilbert	1.03146	0.43715	0.47717	0.67845
Hilbert/First Touch	Hilbert/First Touch	0.47147	0.19453	0.35931	0.52089

Table 4: Results of the Different Data and Computation Reorderings for *Magi*
(Ratios as Compared to the Original Measurements)

6. Future Work

Thus far, we have experimented with two applications and developed a number of strategies for improving performance. We intend to encapsulate software support for the techniques we considered into libraries that will assist with applying data and computation reordering to other applications. As we gain more experience, we will investigate how compilers and tools can help automate application of data and computation reordering strategies. Also, we plan to investigate how hardware performance counters can be used to guide application of periodic data and computation restructuring during long-running computations.

One limitation of the dense "matrix of bins" representation that we are currently using for blocking the computation order of the interactions in *moldyn* is that its size grows quadratically with the total size of the particle data. Very large numbers of particles, very small primary caches, or both can cause the matrix of bins data structure to consume large amounts of storage. At the cost of multiple passes through the interaction list, a standard linear lexicographic sort would reduce the space requirements to a linear factor of the number of blocks. This may prove a better alternative where the L1 cache is extremely small and direct mapped as in the DEC Alpha 21164.

7. Summary and Conclusions

Typically, irregular applications make poor use of memory hierarchies and performance suffers as a result. Improving memory hierarchy utilization involves improving reuse at multiple levels, typically including TLB and one or more levels of cache. In this paper, we have described how a combination of data and computation reordering can dramatically reduce cache and TLB miss rates and improve memory bandwidth utilization. We have shown that neither data reordering nor computation reordering alone is nearly as effective as the combination. We introduced multi-level blocking as a new computation reordering strategy for irregular applications and demonstrated significant benefits by combining reordering techniques based on space-filling curves with other data or computation reordering techniques.

Using space-filling curves as the basis for data and computation reordering offers several benefits. First, reordering data elements according to their position along a space-filling curve probabilistically increases spatial locality. In space-filling curve order, neighboring elements in physical space, which tend to be referenced together during computation, are clustered together in memory. This clustering helps improve utilization of long cache lines and TLB entries. Second, reordering computation to traverse data elements in their order along a space-filling curve also improves temporal locality. By following the space-filling curve, neighboring elements in physical space are processed close together in time, and thus computations that operate on a data element and its spatial neighbors repeatedly encounter the same elements as the computation traverses all elements in a neighborhood. Finally, data reordering based on position along a space-filling curve is fast. The cost of such a reordering is typically small relative to the rest of a program's computation.

With the *moldyn* application, we demonstrated dramatic improvements in memory hierarchy utilization and cut overall execution time in half by composing a data reordering based on a space-filling curve with a computation reordering that is blocked for the memory hierarchy. The multi-level blocking algorithm's efficient temporal reuse of data complements the improved spatial locality resulting from the space-filling curve data ordering.

With the *magi* application, composing computation reordering based on space-filling curves with a first-touch data reordering strategy improved memory hierarchy performance and reduced execution time better than either strategy alone. The improved temporal locality achieved by considering particles in space-filling curve order combined with the improved spatial locality provided by first touch improves reuse at all levels of the memory hierarchy and reduces execution time by 5% more than the best single strategy. Overall the combination of techniques cut the execution time nearly in half.

As the gap between processor and memory speeds continues to grow and large-scale scientific computations continue their shift towards using adaptive and irregular structures, techniques for improving the performance of memory hierarchy for irregular applications will become increasingly important. While in the future it is likely that compilers and tools will provide significant help in applying data and computation reordering, our experience leads us to believe that determining the legality of computation reordering for irregular codes will require user assistance.

8. References

[1] David Callahan, Steve Carr, and Ken Kennedy, "Improving Register Allocation for Subscripted Variables," *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design & Implementation*, pp. 53-65 (June 1990).

[2] D. Gannon, W. Jalby, and K. Gallivan, "Strategies for Cache and Local Memory Management by Global Program Transformation," *Journal of Parallel and Distributed Computing* **5** pp. 587-616 (1988).

[3] M. E. Wolf and M. S. Lam, "A Data Locality Optimizing Algorithm," *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pp. 30-44 (June 1991).

[4] A. K. Porterfield, *Software Methods for Improvement of Cache Performance on Supercomputer Applications,* PhD Dissertation, Rice University, Houston, TX (May 1989).

[5] J. Ferrante, V. Sarkar, and W. Thrash., "On Estimating and Enhancing Cache Effectiveness," *Proceedings of Fourth Workshop on Languages and Compilers for Parallel Computing*, (Aug 1991).

[6] D. M. Tullsen and S. J. Eggers, "Effective cache prefetching on bus-based multiprocessors," *ACM Transactions on Computer Systems* **13**(1) pp. 57-88 (February 1995).

[7] Todd C. Mowry, Monica S. Lam, and Anoop Gupta, "Design and Evaluation of a Compiler Algorithm for Prefetching," *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 62-73 (October 1992).

[8] A.C. McKeller and E.G. Coffman, "The Organization of Matrices and Matrix Operations in a Paged Multiprogramming Environment," *Communications of the ACM* **12**(3) pp. 153-165 (1969).

[9] W. Abu-Sufah, D. J. Kuck, and D. H. Lawrie, "Automatic program transformations for virtual memory computers.," *Proceedings of the 1979 National Computer Conference*, pp. 969-974 (June 1979).

[10] S. Carr, *Memory Hierarchy Management,* Ph.D. Dissertation, Rice University, Houston, TX (May 1993).

[11] J.J. Navarro, E. Garcia, and J.R. Herrero, *Proceedings of the 10th ACM International Conference on Supercomputing (ICS)*, (1996).

[12] Induprakas Kodukula, Nawaaz Ahmed, and Keshav Pingali, "Improving Register Allocation for Subscripted Variables," *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design & Implementation*, pp. 346-357 (June 1997).

[13] J. R. Allen and K. Kennedy, "Automatic loop interchange," *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction SIGPLAN Notices* **19**(6) pp. 233-246 (June 1984).

[14] K. S. McKinley, S. Carr, and C.-W. Tseng, "Improving Data Locality with Loop Transformations," *ACM Transactions on Programming Languages and Systems* **18**(4) pp. 424-453 (July 1996).

[15] Chen Ding and Ken Kennedy, "Improving Cache Performance of Dynamic Applications with Computation and Data Layout Transformations," *To appear in Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design & Implementation*, (May 1999).

[16] Hans Sagan, *Space-Filling Curves,* Springer-Verlag, New York, NY (1994).

[17] Hanan Samet, *Applications of Spatial Data Structures: Computer Graphics, Image Processing and GIS,* Addison-Wesley, New York, NY (1989).

[18] Michael S. Warren and John K. Salmon, "A Parallel Hashed Oct-Tree N-Body Algorithm," *Proceedings of Supercomputing '93*, (November 1993).

[19] Chao-Wei Ou, Manoj Gunwani, and Sanjay Ranka, "Architecture-Independent Locality-Improving Transformations of Computational Graphs Embedded in k-Dimensions," *Proceedings of the International Conference on Supercomputing*, (1995).

[20] Manish Parashar and James C. Browne, "On Partitioning Dynamic Adaptive Grid Hierarchies," *Proceedings of the Hawaii Conference on Systems Sciences*, (January 1996).

[21] Mithuna Thottethodi, Siddhartha Chatterjee, and Alvin R. Lebeck, "Tuning Strassen's Matrix Multiplication Algorithm for Memory Efficiency," *Proceedings of SC98: High Performance Computing and Networking*, (November 1998).

[22] Ibraheem Al-Furaih and Sanjay Ranka, "Memory Hierarchy Management for Iterative Graph Structures," *Proceedings of the International Parallel Processing Symposium*, (March 1998).

[23] Donald Knuth, *The Art of Computer Programming Volume 3: Sorting and Searching,* Addison-Wesley, New York, NY (1973).

[24] B.R. Brooks, R.E. Bruccoleri, B.D. Olafson, D.J. States, S. Swaminathan, and M. Karplus, "CHARMM: A Program for Macromolecular Energy, Minimization and Dynamics Calculations," *Journal of Computational Chemistry* **187**(4)(1983).