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
{johnmc,ken}@cs.rice.edu

‡ Computer Science Department
Florida State University
Tallahassee, FL 32306-4530
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 aren't as effective for irregular applications. This paper investigates using data and computation reordering to improve memory hierarchy utilization for irregular applications on systems with multi-level memory hierarchies. We evaluate the impact of data and computation reordering using space-filling curves and introduce multi-level blocking as a new computation reordering strategy for irregular applications. In experiments that applied specific combinations of data and computation reorderings to two irregular programs, overall execution time dropped by a factor of two for one program and a factor of four for the second.

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 [CCK90, GJG88, LRW91, Por89, WoL91, FST91] and data prefetching [Por89, TuE95, MLG92] have significantly improved memory hierarchy utilization for regular applications. A limitation of these techniques is that they aren't as effective for irregular applications. Improving performance for irregular applications is extremely important since large-scale scientific and engineering simulations are increasing using adaptive irregular methods.

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 system 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 memory hierarchy utilization 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 decrease latency and 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 explores strategies for data reordering and computation reordering along with integrated approaches to evaluate how effectively they improve memory hierarchy utilization on machines with multi-level memory hierarchies. We also introduce multi-level blocking as a new computation reordering strategy for irregular applications.

A common class of irregular applications considers particles or mesh elements in spatial neighborhoods. Figure 1 shows a simple 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 over time 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

* This research was supported in part the National Science Foundation under cooperative agreement CCR-9120008, the Department of Energy's Accelerated Strategic Computing Initiative under research subcontract B347884, and by DARPA and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-96-1-0159. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies or endorsements, either expressed or implied, of DARPA and Rome Laboratory or the U.S. Government.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICS '99 Rhodes Greece

Copyright ACM 1999 1-58113-164-x/99/06...$5.00

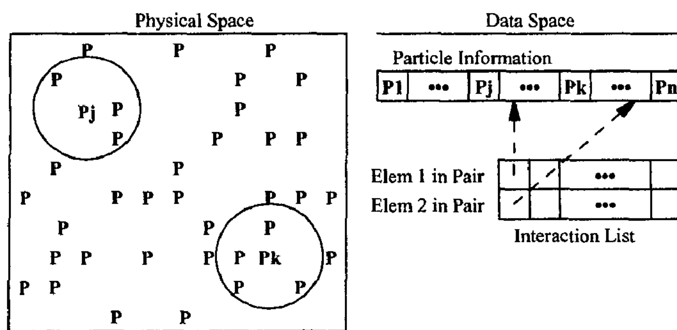

Figure 1: A Classical Irregularly Structured Application

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 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 memory hierarchy performance of applications. 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 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 the last few decades. Early papers focused on blocking to improve paging performance [McC69, AKL79], but recent work has focused more narrowly on improving cache performance [GJG88, WoL91, Por89, FST91]. Techniques similar to blocking have also been effectively applied to improvement of reuse in registers [CCK90]. 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 [NGH96] and a paper by Kodukula *et al.* presents a data-centric blocking algorithm that can be effectively applied to multi-level hierarchies [KAP97].

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 [AlK84, MCT96].

As described earlier, data reordering can be used to reduce bandwidth requirements of irregular applications. Ding and Kennedy [DiK99] 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 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 multi-level 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. A space-filling curve for some finite space of d dimensions ($d \geq 2$) is a continuous, non-smooth curve that passes arbitrarily close to every point. Each point in a d-dimensional space can be mapped to the nearest position along a 1-dimensional space-filling curve by applying a sequence of bit-level logical operations to its d-dimensional coordinates. 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 multi-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]

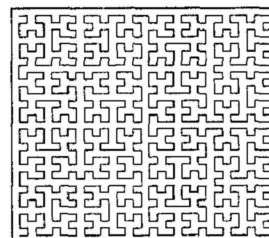

Figure 2: Fifth-order Hilbert curve through 2 dimensions.

Space-filling curves or related ordering techniques [SHT95] have been used to partition data and computation among processors in parallel computer systems. They have been applied in problems domains that include n-body problems [WaS93, SHT95], graph partitioning [OGR95], and adaptive mesh refinement [PaB96]. Ordering 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.* [OGR95] 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.* [TCL98] 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

[1] For more details about the history of space-filling curves, the types of curves, their construction, and their properties, see Sagan [Sag94] and Samet [Sam89].

match for the recursive structure of Strassen's algorithm.

Al-Furaih and Ranka [AlR98] 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% for several computational kernels using 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 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. A data reordering would changes the order of elements within the particle information vector and updates the interaction list to point to the new particle locations. By placing data elements near one another if they are referenced together, 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.

3.1. First Touch Data Reordering

First-touch data reordering is a greedy approach for improving spatial locality of irregular references [DiK99]. 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. An advantage of first-touch data reordering is that the approach is simple and 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.

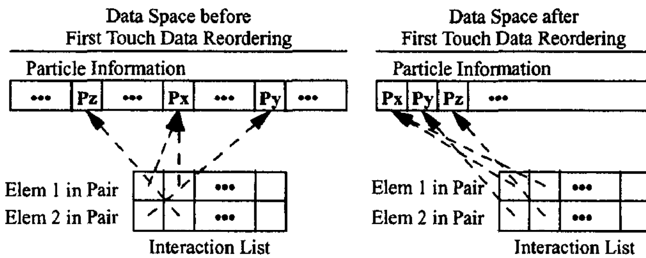

Figure 3: Data Reordering Using a First Touch Approach

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 on the space-filling curve by a sequence of bit-level logical operations. The particles are then sorted into ascending order by their position on the curve. Sorting particles into space-filling curve order tends to increase spatial locality. Namely, if two particles are close together in physical space, then they tend to be nearby 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 access order (e.g. an interaction list), then the access 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.

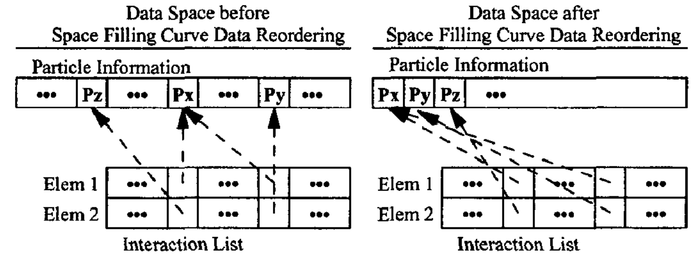

Figure 4: Data Reordering Using a Space Filling Curve

4. Computation Reordering Approaches

A computation reordering involves changing the order in which data elements are referenced, but not the locations in which these data elements are stored. Consider again the data space shown in Figure 1. A computation reordering would reorder the pairs of elements within the interaction list. The vector of particle information accessed by 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 together in time. The following subsections describe the computation reordering approaches considered in this work.

4.1. Space-Filling Curve Computation Reordering

Reordering a 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 these 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

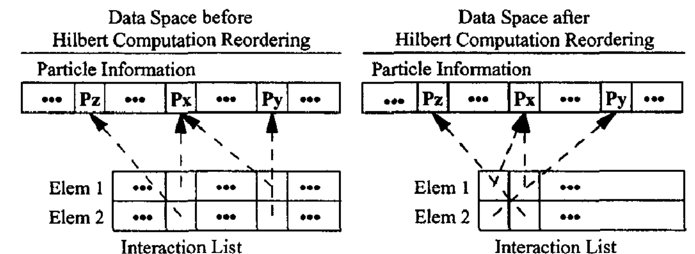

Figure 5: Computation Reordering Using a Space-Filling Curve

information itself remains unchanged. A space-filling curve based computation reordering can improve temporal locality. For instance, if particle X interacts with a nearby particle Y, then it is likely that particle Y will be referenced again soon since Y in turn will interact with other particles.

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 in 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 = i to number of blocks of particles DO
        process interactions between all interacting
        particle pairs with the first particle in block i
        and the second 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, which 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 of memory hierarchy, one would 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.

We extend this strategy to multiple levels by constructing a tuple of block numbers for each interaction pair and then sorting interactions according to their tuple of block numbers. Figure 6 shows how we construct a tuple of block numbers for an interaction of a pair of particles. We first compute an integer tuple of block numbers for each particle, one block number for each level of the memory hierarchy. Next, we interleave the tuples for each of the particles in an interaction pair. Finally, a lexicographic sort [Knu73] on the resulting interleaved tuples achieves the multi-level blocking.[2] This has the effect of first sorting by the smallest block size, corresponding to L1 cache, followed by sorts according to 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.

[2] Lexicographic sort repeatedly reorders tuples by applying an order-preserving radix sort for each element in the tuple from right to left.

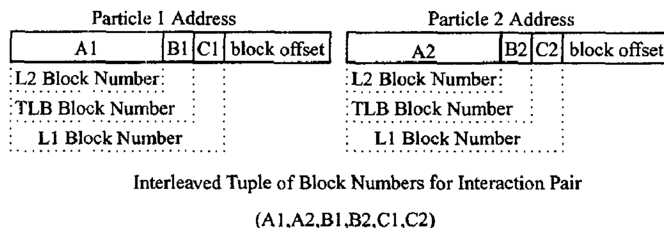

Figure 6: Interleaved Tuple of Block Numbers

5. Applying the Techniques

This section 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 and we were able to use the workstation in isolation. Both programs were compiled with the highest level of optimization available for the native C and Fortran compilers.[3] 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.

Cache Type	Cache Configuration		
	Cache Size	Associativity	Block Size
L1 Data	32KB	2-way	32B
L2 Data	1MB	2-way	128B
TLB	512KB	64-way	8KB

Table 1: SGI O2 Workstation Cache Configurations

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 [BBO83], and closely resembles the structure represented in Figure 1 of the paper. An interaction 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.

[3] 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 [MLG92]. 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.

A high-level description of the computation for *moldyn* is shown in Figure 7. The time-consuming portion of the algorithm is the inner **FOR** loop which corresponds to 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 reordering 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 7: 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 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 8 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.

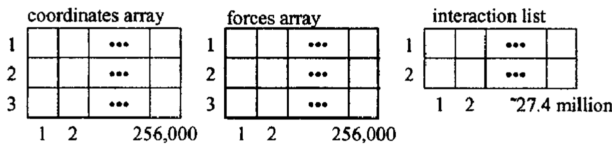

Figure 8: Main Data Structures in the *Moldyn* Benchmark

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 as possible interaction partners. (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.

(3) We adjusted the loop that computes forces so that when a sequence of interactions references the same first particle, the data for the first particle is only loaded from memory once.

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 version of the program that we refer to as *Baseline*.

Table 2 shows information about misses in the caches and the TLB for our Baseline version of *moldyn* benchmark. To investigate the nature of the poor memory hierarchy performance, we used a simulator to collect an L1 miss trace for the application. Figure 9 shows a plot of L1 misses over the first 100,000 interactions within the *computeforces* in the Baseline version of *moldyn*. While all memory references were simulated, only the misses associated with the particle information are displayed in the plot. In the plot the block numbers are the portion of the addresses (tag and index) used to access the L1 cache and the interaction numbers indicate on which interaction each miss occurred. The band of misses is initially as wide as the array of particles. The lower border of this band slowly rises as the interaction numbers increase since a particle only has interactions with higher number particles. Figure 10 shows a plot of L1 misses over 100,000 interactions when a Hilbert curve was used to reorder the particle data and blocking was used to reorder the interactions. This plot was drawn at the same scale as the plot in Figure 9 and the total number of misses for the first 100,000 interactions was reduced by a factor of 25. The difference between these plots illustrates the dramatic performance benefits that can be achieved by applying data and computation reorderings.

Cache Type	Baseline Misses	Baseline Miss Ratio
L1	1,613,065,560	0.23439
L2	995,152,174	0.61693
TLB	664,457,217	0.09655

Table 2: Miss Information

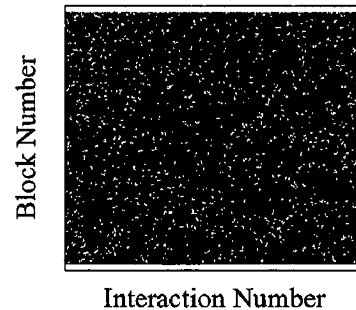

Figure 9: L1 Baseline Misses over the First 100,000 Interactions

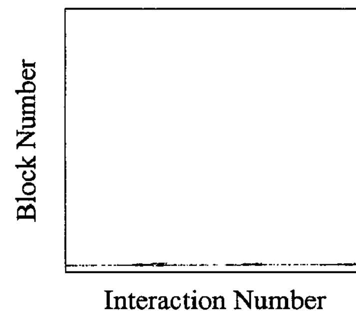

Figure 10: L1 Misses over the First 100,000 Interactions after Reordering the Data in Hilbert Order and Blocking the Computation

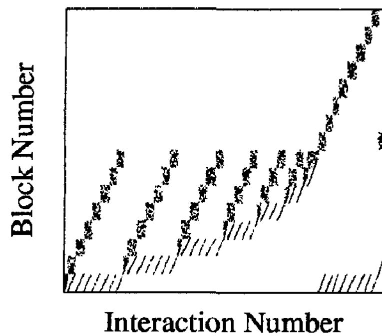

Figure 11: Plot of 10K L1 Misses

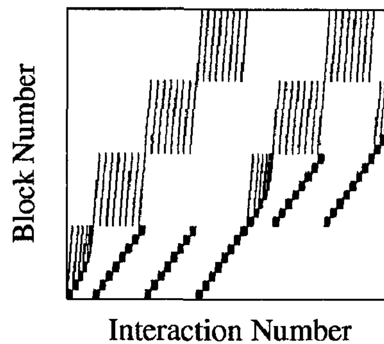

Figure 12: Plot of 100K L1 Misses

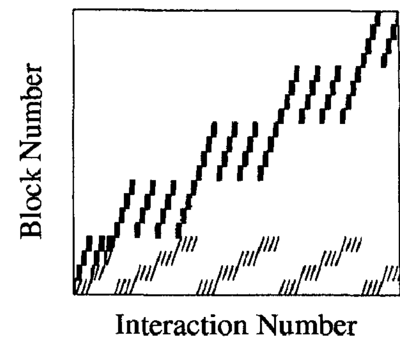

Figure 13: Plot of 1M L1 Misses

Data Reordering	Computation Reordering	L1 Cache Misses	L2 Cache Misses	TLB Misses	Cycles
First Touch	None	0.87487	0.76548	0.31928	0.79069
Hilbert	None	0.87978	0.78074	0.26397	0.80731
None	Hilbert	0.45053	0.12157	0.74006	0.37778
None	Blocking	1.26245	0.45723	0.20846	0.63187
First Touch	Hilbert	0.33735	0.14314	0.00806	0.38773
Hilbert	Hilbert	0.25816	0.10139	0.00624	0.26550
Hilbert	Blocked	0.25016	0.11936	0.00626	0.30260

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

To accomplish multi-level blocking of the *moldyn* non-bonded forces computation, the interaction list must be reordered to match the characteristics of the memory hierarchy of the target machine. As described in Section 4.2, we can achieve such a blocking by lexicographically sorting interaction pairs according to the interleaved tuples of block numbers we compute for each pair. To achieve the effect of this sort quickly in practice, we concatenate the tuple of block numbers in sequence to form a composite block number, break each composite block number into sections of K (or fewer) bits to form a refined tuple,[4] and then lexicographically sort based on the refined tuple. Using K=20 on an SGI O2 workstation, we were able to efficiently sort the entire 27 million pair interaction list for *moldyn* using a single radix sort.

We found through experimentation that good blocking factors for *moldyn* on an SGI O2 are approximately one half the cache size for the L1 and L2 caches and one quarter the TLB size for the TLB. To show how our multi-level blocking algorithm regularizes memory accesses, we include Figures 11-13 which show the pattern of L1 misses due to the *computeforces* function for the first 10,000, 100,000, and 1,000,000 misses, respectively, when no data reordering is performed and *moldyn* is blocked for an SGI O2 workstation. Note that the scales of these three plots differ on each axis. These plots show only the misses in the particle information. All interactions within an L1 block are processed before proceeding to 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. Figure 11 shows misses across eight blocks of particle information. These eight blocks are repeatedly accessed, which illustrates that a TLB block is eight times the size of a L1 block. Figures 12 and 13 plot enough L1 misses to distinguish interactions between TLB blocks and L2 blocks, respectively. Figure 13 illustrates that four TLB blocks are accessed repeatedly since there are four TLB blocks for each L2 block.

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 execution of the Baseline version of *moldyn* without any run-time 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 first touch data reordering was applied first, then blocking would change the order of the references and the benefits from the first-touch ordering would be diminished. 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, data and computation reordering are most effective at reducing misses for caches with a large block or line size. For this reason reductions in TLB misses were the greatest, and those for L2 were greater than those for primary cache. Second, a combination of data and computation reorderings performed dramatically better than using any specific type of data or computation reordering in isolation. Hilbert data reordering combined with either Hilbert computation reordering or computation reordering based on multi-level blocking reduced TLB misses by a factor of 160, L2 misses by a factor of 10, and primary cache misses by a factor of 4. This strategy reduced the miss ratios for L1 cache from 23.4% to 6.1%, for L2 cache from 61.7% to 6.3%, and for

[4] K should be chosen so that a single radix sort would not be too space inefficient or thrash the memory hierarchy.

TLB from 9.7% to 0.06%.[5] In terms of reducing execution cycles, Hilbert-based data and computation reordering performed the best, yielding a factor of four overall reduction in cycles. While the blocking strategy was competitive, the Hilbert-based data and computation reordering does so well for this interaction density that the modest additional reduction in misses obtained by multi-level blocking did not fully amortize the overhead of performing the blocking, which required 8.5% of the Baseline cycles. If the overhead of blocking were amortized over a greater number of time steps, then its superior memory hierarchy utilization would make blocking the most effective computation reordering.

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 14.

```
Read in the data for 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 14: 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 containing 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 interaction lists in *magi* are established at the beginning of each time step, which causes the first-touch data reordering to affect the computation order.

Table 4 shows the results of applying 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* did not improve performance. The cost of accessing data through an indirection vector offset 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.

The results in Table 4 show that the combination of reordering particle data and and interaction computations according to particle positions along a Hilbert curve (which probabilistically increase spatial and temporal locality) followed by a first-touch data reordering (which greedily improve spatial locality) achieves the lowest L2 and TLB misses and the best overall cycle time by a very slim margin. The table shows that applying a first-touch data reordering after the Hilbert-based reordering amortizes the cost of the first-touch reordering by reducing L2 and TLB misses, but the barely perceptible improvement in overall

[5] 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 the rest of the program execution. When we consider the performance of the *computeforces* routine alone, improvements are far greater.

Data Reordering	Computation Reordering	L1 Cache Misses	L2 Cache Misses	TLB Misses	Cycles
First Touch	First Touch	0.42959	0.27032	0.49173	0.56321
Hilbert	Hilbert	0.28621	0.11916	0.15704	0.43751
Hilbert/First Touch	Hilbert/First Touch	0.32670	0.11695	0.13513	0.43607

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

performance does not justify the additional programming effort.

6. Status and Future Work

We have encapsulated our multi-level blocking technique for reordering computation into a flexible library routine that uses a variable length vector of blocking factors to block computations for a multi-level memory hierarchy of arbitrary depth. We have used this library to block *moldyn* for the DEC Alpha 21164A with results similar to those we presented for the SGI O2.

Thus far, we have experimented with two applications and developed a number of performance improving strategies. 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.

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 by using Hilbert-based data reordering and either multi-level blocking or a Hilbert-based strategy for reordering computation. Our experiments showed that while the multi-level blocking algorithm had lower miss ratios at all levels of the memory hierarchy for the computation kernel, its end-to-end performance was slightly worse. The higher cost in end-to-end cycles is due to the cost of blocking the computation. The Hilbert-based computation reordering has an advantage in that it is accomplished at no cost by simply performing Hilbert-based data reordering before building the interaction list in the canonical fashion. The benefits of blocking would be larger with a higher density of interactions per particle. For very high interaction densities, the careful scheduling of interactions by multi-level blocking could achieve significant temporal reuse even when the Hilbert-based computation reordering would not (i.e. if there are more interaction partners per particle than would fit in L1 cache). 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, Hilbert curve based strategies for data and computation reordering improved end-to-end performance by over a factor of two. The best memory hierarchy utilization came from considering particles in space-filling curve order to improve temporal locality, and using that as the basis for a first-touch data and computation reordering that greedily improves spatial locality. It is interesting to note that the improvements we achieved for *magi* with our data reorderings are relative to a baseline computation for which input particle data has already been carefully ordered using a relabeling algorithm by Scott Sloan of the University of Newcastle.

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 memory hierarchy performance of irregular applications will become increasingly important. Although our experience leads us to believe that determining the legality of computation reordering will require user assistance, once that knowledge is available, future compilers should be able to generate the run-time code to automatically produce the reorganizations described here.

8. Acknowledgements

Vikram Adve and Rob Fowler offered their insights in discussions that helped shape this work. Doug Moore provided us with a library for mapping between multidimensional coordinate spaces and positions along a Hilbert space-filling curve. Gina Goff and Ehtesham Hayder provided us access to an SGI O2 workstation for our experiments. Initial experimentation for this work was performed on an SGI Onyx2 at the Biomedical Computation and Visualization Laboratory of the Baylor College of Medicine supported by grant BIR-9512521 from the National Science Foundation.

9. References

[AKL79] 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 (Jun 1979).

[AlR98] I. Al-Furaih and S. Ranka, "Memory Hierarchy Management for Iterative Graph Structures," *Proceedings of the International Parallel Processing Symposium*, (Mar 1998).

[AlK84] 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 (Jun 1984).

[BBO83] 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).

[CCK90] D. Callahan, S. Carr, and K. Kennedy, "Improving Register Allocation for Subscripted Variables," *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design & Implementation*, pp. 53-65 (Jun 1990).

[DiK99] C. Ding and K. 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).

[FST91] 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).

[GJG88] 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).

[Knu73] D. Knuth, *The Art of Computer Programming Volume 3: Sorting and Searching,* Addison-Wesley, New York, NY (1973).

[KAP97] I. Kodukula, N. Ahmed, and K. Pingali, "Data-centric Multi-level Blocking," *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design & Implementation*, pp. 346-357 (Jun 1997).

[LRW91] M. S. Lam, E. E. Rothberg, and M. E. Wolf, "The Cache Performance and Optimizations of Blocked Algorithms," *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 63-74 (Apr 1991).

[McC69] 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).

[MCT96] 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 (Jul 1996).

[MLG92] T. C. Mowry, M. S. Lam, and A. 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 (Oct 1992).

[NGH96] J. J. Navarro, E. Garcia, and J. R. Herrero, *Proceedings of the 10th ACM International Conference on Supercomputing (ICS)*, (1996).

[OGR95] C. Ou, M. Gunwani, and S. Ranka, "Architecture-Independent Locality-Improving Transformations of Computational Graphs Embedded in k-Dimensions," *Proceedings of the International Conference on Supercomputing*, (1995).

[PaB96] M. Parashar and J. C. Browne, "On Partitioning Dynamic Adaptive Grid Hierarchies," *Proceedings of the Hawaii Conference on Systems Sciences*, (Jan 1996).

[Por89] A. K. Porterfield, *Software Methods for Improvement of Cache Performance on Supercomputer Applications,* PhD Dissertation, Rice University, Houston, TX (May 1989).

[Sag94] H. Sagan, *Space-Filling Curves,* Springer-Verlag, New York, NY (1994).

[Sam89] H. Samet, *Applications of Spatial Data Structures: Computer Graphics, Image Processing and GIS,* Addison-Wesley, New York, NY (1989).

[SHT95] J. P. Singh, C. Holt, T. Totsuka, A. Gupta, and J. Hennessy, "Load Balancing and Data Locality in Adaptive Hierarhcical N-body Methods: Barnes-Hut, Fast Multipole, and Radiosity," *Journal of Parallel and Distributed Computing*, (Jun 1995).

[TCL98] M. Thottethodi, S. Chatterjee, and A. R. Lebeck, "Tuning Strassen's Matrix Multiplication Algorithm for Memory Efficiency," *Proceedings of SC98: High Performance Computing and Networking*, (Nov 1998).

[TuE95] D. M. Tullsen and S. J. Eggers, "Effective cache prefetching on bus-based multiprocessors," *ACM Transactions on Computer Systems* **13**(1) pp. 57-88 (Feb 1995).

[WaS93] M. S. Warren and J. K. Salmon, "A Parallel Hashed Oct-Tree N-Body Algorithm," *Proceedings of Supercomputing '93*, (Nov 1993).

[WoL91] 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 (Jun 1991).