Software Support For Improving Locality in Advanced Scientific Codes

Chau-Wen Tseng

Department of Computer Science
University of Maryland
College Park, MD 20742
tseng@cs.umd.edu

Abstract

Programs can achieve good performance only if they possess *data locality*, This paper describes our proposal to develop and evaluate software support for improving locality for advanced scientific applications for both sequential and parallel machines. The basic premise is that both compile-time analyses and sophisticated run-time systems are necessary. Run-time systems are needed because many programs are not analyzable statically. Compiler support is crucial both for inserting interfaces to the run-time system and for directly applying program transformations where possible. We examine locality optimizations needed for three features of advanced scientific applications (3D arrays, irregular accesses, and pointers). Preliminary experimental evaluation is very encouraging, but much work remains to automate and improve these compiler and run-time systems. We propose to extend locality optimizations in several directions, handling: cache conflicts between multiple data, deep memory hierarchies (multi-level caches and TLBs), hardware and software prefetching, nonlinear memory layouts, parallel and cluster architectures, and memory performance tools. Because of trends in computer architectures, lessons learned here are also likely to prove very useful for other application domains, including image processing and high-performance databases.

Technical Report CS-TR-4168 (also UMIACS-TR-2000-56), Dept. of Computer Science, University of Maryland, June 2000

1 Introduction

From astrophysics to biochemistry, scientists are increasingly using computers to conduct research and development. Fueling the trend is the wide availability of inexpensive commodity microprocessors and soon multiprocessor PCs. Though computers are quite fast, scientists continue to seek greater computing power to perform ever more complex simulations. Parallelism has always been accepted as a technique for greatly improving available computation power, at least for amenable applications. In recent years, *locality* has joined parallelism as a key ingredient to achieving high performance, due to the increasing disparity between memory and processor speeds.

To remain successful, computers not only need to be fast, but easy to use. Their computing power should be accessible without expert knowledge of the underlying architecture. Software support at both compile and run time plays an important role by automatically customizing programs for complex processor architectures, improving portability, and providing high performance to non-expert programmers. Compiler transformations for improving data locality have proven to be quite successful, either rearranging computation through loop transformations (e.g., loop permutation, fusion, fission) [57, 87], or changing data layout through data transformations (e.g., padding, transpose) [3, 39, 69]. Experiments show compilers can improve the performance of many scientific programs, some times dramatically.

Despite these successes, new problems are beginning to arise. Scientific codes are becoming increasingly large and complex as more difficult multi-disciplinary problems are attempted. Features such as 3D arrays, adaptive irregular memory accesses, and pointer-based data structures make utilizing caches effectively difficult, leading to poor performance. Traditional program transformations are frequently inapplicable or insufficient.

We believe exploiting locality for advanced scientific applications will require developing novel compile-time analyses and sophisticated run-time systems to perform data layout and computation transformations. Run-time systems are needed because many programs are not analyzable statically, while compiler support is still crucial both for inserting interfaces to the run-time system and for directly applying program transformations where possible. Cooperation between the compiler and run-time will be critical for advanced scientific codes.

This proposal aims to develop and evaluate software support for improving locality for advanced scientific applications, for both sequential and parallel machines. To illustrate how compiler and run-time software support can be combined to improve locality, we examine problems and preliminary solutions for three types of scientific codes (3D iterative solvers, irregular computations, and pointer-based data structures) on both sequential and parallel computers, then present our proposed research areas. We are implementing our optimizations in COSMIC, an optimizing compiler based on the Stanford SUIF parallelizing compiler [28, 86]. We begin by quickly reviewing some basic concepts regarding caches and locality, as well as existing locality optimizations.

1.1 Caches and Locality

The rise in the importance of data locality is driven by the growing disparity between processor and memory speeds. As processers become ever faster relative to memory, computer architects have created in modern processors a *memory hierarchy* where higher levels (e.g., registers, primary cache) are fast and small, while lower levels (e.g., secondary cache, memory) are slow and large. For a more concrete example of how memory latencies affect performance, consider recent microprocessors such as the DEC Alpha or the MIPS R10000. Data in primary (on-chip) cache can be accessed in a few cycles, secondary (off-chip) cache accesses take 10–20 cycles, and local memory accesses take 60–100 cycles. Processors can now issue multiple instructions per cycle, making long memory latencies even worse obstacles to good performance. Applications that do not use caches effectively spend most of their time waiting for memory accesses.

Because of hardware limitations, caches are divided into a small number of cache lines. Caches can thus be characterized by the length of their cache lines and capacity (size as determined by the product of cache line size and total number of cache lines). Another important property of caches is that they usually have a small fixed *set associativity*, where a k-way associative cache means an address can be mapped into one of k possible cache lines. Limited associativity means a memory access may be mapped on top of a useful cache line, flushing it from the cache despite sufficient capacity in the cache. These *conflict misses* can cause problems when trying to exploit reuse.

In order to exploit the greater speed of higher levels of the memory hierarchy, applications must possess *data locality*, where an application reuses data already brought in. Reuse may be in the form of *temporal locality*, where the same data is accessed multiple times, or *spatial locality*, where nearby data is accessed together. With respect to the program, locality may be in the form of *self-reuse* caused by a single variable reference, or *group-reuse* caused by multiple variable references. For parallel programs, locality also means accessing data in local memory instead of remote memory on another processor node.

Many compiler techniques exist for improving locality in both sequential and parallel scientific programs. Loop transformations (e.g., loop permutation, fusion, tiling) for sequential dense-matrix codes with regular memory access patterns has proven useful [19, 27, 48, 49, 57, 55, 71, 76, 87, 88]. Data layout optimizations (e.g., transpose, padding) also help [2, 3, 13, 18, 39, 69, 70], even for irregular [1, 22, 58] and pointer-based programs [8, 17]. Despite the major advances made in providing software support for improving locality for both sequential and parallel programs, more work remains for advanced scientific computations.

2 Advanced Scientific Applications

We begin by presenting three types of scientific codes where traditional compiler locality optimizations are insufficient.

2.1 3D Iterative PDE Solvers

An important class of scientific programs computes solutions to systems of partial differential equations (PDEs) using finite differencing techniques. These iterative PDE solvers are also called *stencil* codes because they compute point values based of averaging neighboring points. Historically PDE solvers have targeted 2D domains, but as computers became more powerful scientists began writing programs to solve PDEs in three dimensions. Numerical analysts then discovered that 3D PDE codes have particularly poor memory behavior with respect to microprocessor caches [4, 77, 85].

To see why 3D codes pose greater difficulties for caches, consider the 3D multigrid kernel in Figure 1. Multigrid solvers use a succession of grid sizes to quickly compute solutions to PDEs. Their most computationally expensive part is the smoothing step, which is a simple stencil computation such as Jacobi or red-black successive-over-relaxation (SOR) applied to the component mesh. For instance, over 90% of the execution time in the SPEC95/NAS MGRID benchmark is spent in a 27-point 3D Jacobi kernel.

In a 2D stencil kernel, three adjacent array columns are accessed together. Since adjacent array columns are contiguous in memory, the cache only needs to be able to hold three array columns to exploit all group reuse (assuming a write-through cache, so A does not interfere). Even a relatively small 16K primary (L1) cache can thus hold all the data needed up to a 682xM array (where M can be any value), assuming double precision (8 byte) elements. For 2D arrays of column size 682 or smaller, values of B only need to be brought into cache once for each iteration of the outer time-step loop.

In comparison, the 3D stencil kernel accesses six array columns in three adjacent planes. Because three entire NxN planes now need to remain in cache, only 3D arrays of size 26x26xM can fully exploit reuse for a 16K L1 cache. Even for a larger 512K secondary (L2) cache, group reuse is lost for 3D arrays larger than 147x147xM. Data for the array U will need to be brought into cache two or more times each time the kernel is executed, reducing performance for larger problem sizes.

```
// 3D Multigrid Kernel
R(N,N,N),V(N,N,N),U(N,N,N)
do k=2,N-1
do j=2,N-1
do i=2,N-1
  R(i,j,k) = V(i,j,k)- A *
    (U(i-1,j,k)+U(i+1,j,k)+
     U(i,j-1,k)+U(i,j+1,k)+
     U(i,j,k-1)+U(i,j,k+1))
```

```
// Molecular Dynamics
X(M),Y(M),E(2,N)
do t = 1, time
  if (recalc)
    E(...) = ...
  do i = 1, N
    d = X(E(1,i))-X(E(2,i))
    force = d**(-7)-d**(-4)
    Y(E(1,i)) += force
    Y(E(2,i)) += -force
```

```
// Linked List Traversal
struct node {val, next};
node *ptr, *list;
while (...) {
  ptr->next = malloc(node);
  ptr = ptr->next;
  ptr->val = ... ;
}
while (ptr->next) {
  ...
  ptr = ptr->next;
}
```

Figure 1 Advanced Scientific Codes

2.2 Irregular Computations

Another type of scientific application which underutilizes cache are computations with irregular memory access patterns. These computations arise in several application domains As computational scientists attempt more complex simulations. In computational fluid dynamics (CFD), meshes for modeling large problems are sparse to reduce memory and computations requirements. In n-body solvers such as those arising in astrophysics and molecular dynamics, data structures are by nature irregular because they model the positions of particles and their interactions. Unfortunately, irregular computations have poor temporal and spatial locality, and do not utilize processor caches efficiently.

Consider the example irregular molecular dynamics code in Figure 1, which calculates forces between pairs of atoms in a molecule. Accesses to the index array E are very regular, striding through the array sequentially. However, accesses to arrays X and Y are irregular, since they are dictated by the contents of E. Unlike stencil computations, compile-time transformations cannot improve locality, because the values of index array E are not known at compile time. For these types of codes, irregular memory access patterns mean spatial locality is unlikely to be obtained if the data is larger than the cache.

2.3 Pointer-Based Data Structures

Finally, we also consider programs using recursive data structures. Scientific codes have traditionally relied on arrays of data, and are usually written in languages such as Fortran. The main reason, of course, is performance. The perception (and frequently reality) that pointer-based programs such as C, C++, and Java are inefficient and cannot provide sufficient performance. However, processors are becoming sufficiently fast that ease of programming must also be considered. Compilers are also becoming sufficiently powerful that C and C++ programs written in a Fortran-like style can achieve comparable performance.

As scientific computations become more complex, the ability to dynamically allocate memory and use pointer-based data structures such as linked lists, binary/quad/oct trees, directed acyclic graphs (DAGs), and other graph structures becomes very useful to scientists. The role of compilers and run-time systems then becomes trying to keep the cost of using pointer-based data structures from becoming prohibitive.

One of the main disadvantages of pointer-based data structures is their lack of locality. To see why, consider the example list creation and traversal code in Figure 1. When pointer-based data structures (e.g., lists, trees) are created, new elements are allocated incrementally with functions such as *malloc*(), with no guarantee of proximity in memory to previous items. Without spatial locality, when the linked list or tree is traversed each new item may cause a cache or TLB miss. It thus makes sense to consider software support for improving the locality of pointer-based data structures, particularly the high-level data structures (lists, trees, DAGs) that are likely to be used in scientific computations.

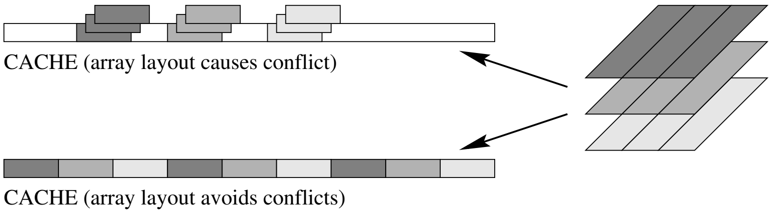


Figure 2 Example of conflict misses under two array layouts

3 Locality Optimization Techniques

Now that we have seen examples of advanced scientific computations which do not utilize caches well, we present some locality optimization techniques we recently developed, as well as evidence of their effectiveness.

3.1 Conflict-Conscious Tiling and Padding

Tiling (blocking) is a transformation which combines strip-mining with loop permutation to form small tiles of loop iterations which are executed together to exploit data locality [11, 37, 89]. Tiling is very effective with linear algebra codes [19, 48, 49, 51, 71], but tiling for iterative PDE solvers is a relatively new area. One reason is that when loops are permuted to exploit temporal and spatial reuse [57, 87], group-temporal and spatial reuse in 2D stencils can usually be obtained without tiling, though padding may be necessary to preserve group reuse [55, 70]. As we have already seen, this is no longer the case for 3D stencil codes. Tiling can be used to exploit reuse across outer time-step loops [76], but this is not possible in multigrid codes because each time step contains sweeps over a sequence of grids of different sizes.

We recently demonstrated tiling can be used to preserve group-reuse and spatial locality in 3D iterative PDE solvers, especially when combined with array padding, a data transformation which increases array dimensions to avoid intra-tile cache conflicts [73]. Our algorithm is as follows. We apply tiling to all three loops, creating tiled code which executes a chunk of the original code at a time, allowing data accessed by the tile to more likely remain in cache. We use a cost model to estimates the reuse gained and lost for different tile dimensions [87].

Algorithm Limited cache associativity may cause conflict misses, severely degrading the performance of tiled codes [51]. For instance, choosing cubic tiles equal in size to the L1 cache (the TILE algorithm evaluated later) often results in self-interfering tiles, as shown in Figure 2. Two key steps of our algorithm are thus calculating tile dimensions and array padding size to avoid cache conflicts. An algorithm (LRW) by Lam, Rothberg, and Wolf finds the largest square nonconflicting 2D tile using an greedy algorithm which expands tile dimensions while checking that no conflicts occur [51]. We can modify this algorithm for three dimensions by introducing the Euclidean algorithm for computing the greatest common divisor [71]. We iteratively expand the third tile dimension, applying the Euclidean algorithm to compute maximal 3D tile sizes for the given array size. Running time is negligible since the Euclidean algorithm requires only a logarithmic number of steps.

Even when choosing nonconflicting tiles, performance of tiling may suffer for certain array dimensions, particularly powers of two. In such cases, non-conflicting tiles result where one or more dimensions are pathologically small, reducing the benefits of tiling. One possible solution is to use padding to enable better tile sizes [3, 69]. Padding increases the size of leading array dimensions, increasing the range of non-conflicting tile shapes. It has proven to be very useful for improving tiling for 2D linear algebra codes [71]. To combine padding with tile size selection, we need only test a small set of pads. Our algorithm (PAD) increases each lower dimension, generating tile sizes for each padded array. We pick a combination tile size and array padding when we are within a few percent of the minimal cost, according to our cost model. In our experiments the additional memory required for padding is quite small, usually

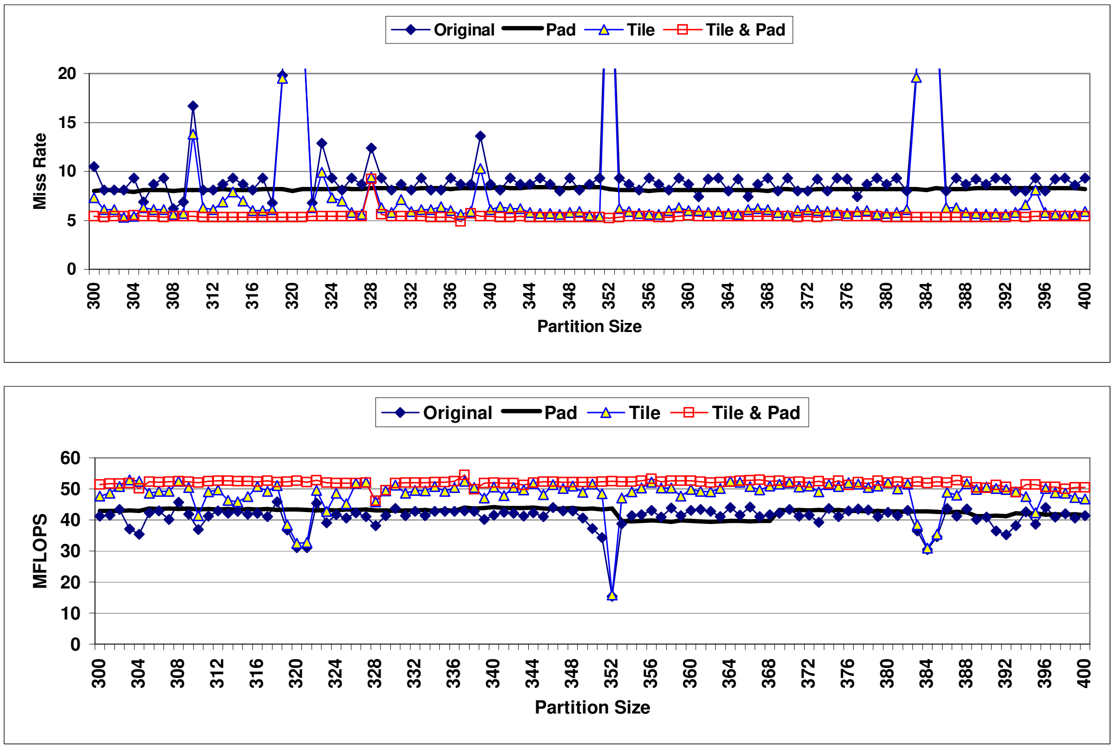


Figure 3 RESID Cache Miss Rates & MFLOPS (16K, direct-mapped cache)

L1 Cache Miss Rate				MFLOPS (% Improvement)			
Orig	Pad	Tile	Tile & Pad	Orig	Pad	Tile	Tile & Pad
10.1%	8.2%	8.3%	5.4%	40.8	42.5 (4%)	48.4 (16%)	53.0 (23%)

Table 1 Optimizations Averaged Over Problem Sizes 300–400 (Ultrasparc2)

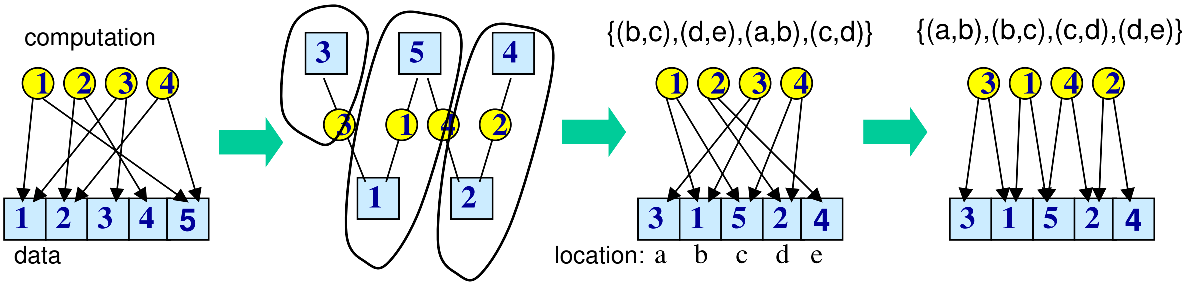
less than 5%.

Evaluation To examine the benefits of tiling 3D PDE solvers, we evaluated our transformations on RESID, a 3D stencil kernel comprising 60% of the execution time of MGRID, a SPEC/NAS benchmark program. We measured both cache miss rates and actual program performance on a 360 MHZ Sun Ultrasparc2 workstation. To compare original and optimized performance thoroughly, we varied problem sizes over a range of values when obtaining performance improvements and cache miss rate rate data. Problem sizes NxNx30 were used, where N ranged from 200 to 400. The third dimension was fixed at 30 to reduce measurement times and had no impact on tile conflicts.

Results are shown in Figure 3 and summarized in Table 1. We see that tiling improves performance but still suffers from excessive cache conflicts for many problem sizes. Padding alone does not improve performance, but eliminate excessive conflict misses at problematic array sizes. Combining tiling and padding achieves both stability and improved performance for 3D stencil codes.

3.2 Run-time Data and Computation Transformations

3D multigrid codes are amenable to compile-time analysis. Recently, researchers found data and computation transformations can also improve the locality of both irregular computations [1, 20, 22, 58, 59] and even programs with pointer-based data structures [8, 16, 17]. These transformations may be applied because irregular scientific computa-



Data Reordering **Computation Reordering**

Figure 4 Overview of Data and Computation Transformations

tions are typically composed of loops performing reductions such as SUM and MAX which can be safely reordered [20, 22]. Many irregular scientific codes access two or more distinct irregular references on each loop iteration. Such codes arise in PDE solvers traversing irregular meshes or N-body simulations, when calculations are made for pair of points. In such cases loop iterations may be considered edges connecting the data accessed on that iteration.

Figure 4 shows an example of data/computation reordering for these computations. Circles represent computations (loop iterations), squares represent data, and arrows represent data accesses. Notice how computations can be viewed as edges connecting data nodes, resulting in a graph. Data reordering partitions the graph and puts all nodes in a partition nearby in memory to improve spatial locality. Computation reordering lexicographically sorts loop iterations then brings accesses to memory together, improving temporal locality.

Algorithm Choosing how to reorder data and computation is the key to improving locality for irregular scientific applications. Sorting loop iterations according to locations of data accesses can yield near-optimal locality for simple computations [59], but data reordering is usually also needed. One heuristic is *consecutive packing* (CPACK), where data is moved into adjacent locations in the order they are first accessed (i.e., first-touch) by the computation [22]. Reverse Cuthill-McKee (RCM) methods using reverse breadth-first search (BFS) can also reorder data with low overhead [20, 1]. Data can also be laid out in memory according to space-filling curves (e.g., Hilbert, Morton) which attempt to minimize the distance (in memory) between two geometrically close points in space [34, 58, 75].

Data and computation can also be *partitioned*, exploiting the fact most interactions (and mesh connections) in scientific computations are local, between nearby elements. Recursive coordinate bisection (RCB) uses geometric coordinate information to recursively split partitions in two along the longest dimension, by finding the median data point in that dimension. Multi-level graph partitioning algorithms such as METIS [46, 41, 42] partition data and computation directly treating it as a graph. Overhead is reduced by computing a succession of smaller graphs which approximate the previous graph and partitioning the coarsened graph.

Both RCB and METIS produce good partitions but with relatively high overheads, making them less suitable for locality optimizations, particularly adaptive codes. To reduce overhead, we recently developed a graph-based partitioning algorithm based on graph coarsening (GPART) [31]. It performs a number of passes, clustering adjacent nodes into a single partition on each pass. Each partition is then treated as a single node in the next pass, creating a hierarchical structure. Overhead is much less than other partitioning algorithms, since for locality optimizations we can halt after a small fixed number of coarsening passes, regardless of the data size.

Evaluation To evaluate locality transformations, we ran a number of experiments on a SGI Origin2000 with R10000 processors and a DEC multiprocessor with Alpha 21064 processors. We used MOLDYN, a program abstracted from the non-bonded force calculation in CHARMM, a key molecular dynamics application used at NIH to model macromolecular

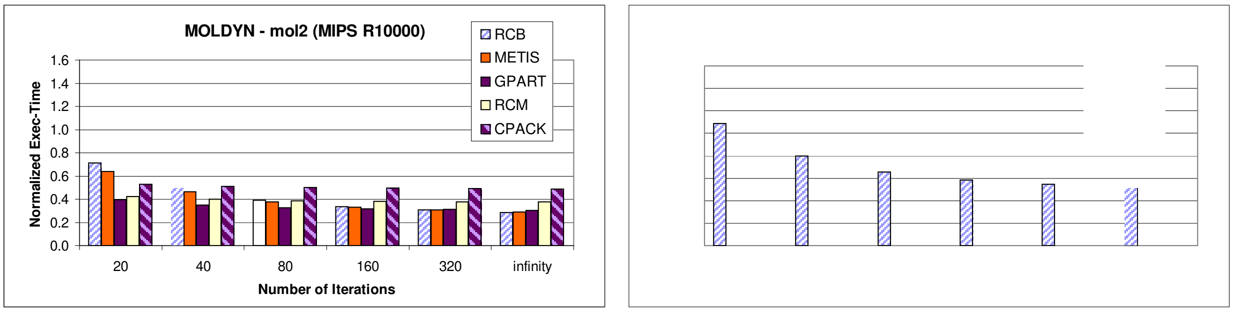


Figure 5 Impact on Execution Time (Overhead Included)

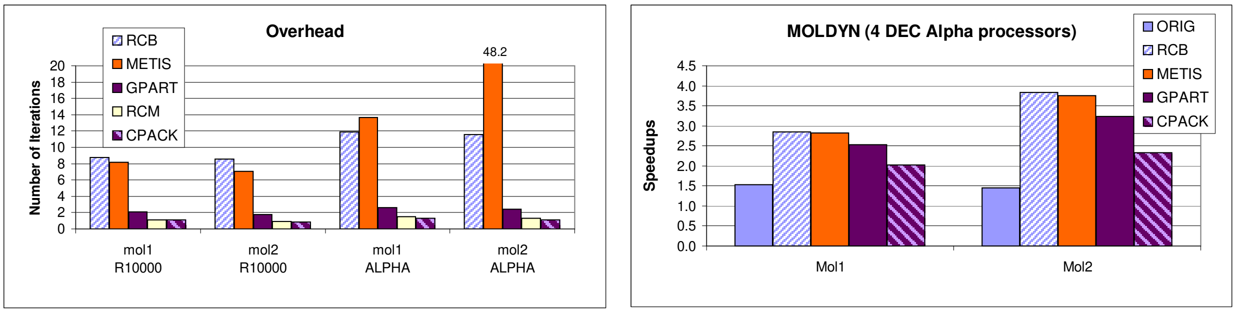


Figure 6 Overhead of Optimizations, Impact on Parallel Execution Time

systems. MOLDYN maintains an edge list representing interactions between pairs of molecules.

Figure 5 presents the sequential execution times for each locality optimization on both the MIPS R10000 and DEC Alpha 21064 varying total number of iterations executed. We found locality transformations can improve the execution time of long-running irregular scientific programs from 50–70% on the R10000 and 30–50% on the Alpha, with benefits increasing with large input data sets. RCB and METIS achieve the best performance, but also have the highest overheads, reducing their benefits for programs executing fewer time step iterations. The improvements from GPART nearly match that of the best partitioning algorithms, but with much lower overhead.

The first graph in Figure 6 shows overhead of each transformation for each input mesh, relative to the execution time of a single time step. Overhead is high for RCB and METIS, moderate for GPART, and low for RCM and CPACK. For short running programs, GPART, RCM, and CPACK will be proper choices. RCB and METIS will achieve more improvements for long running programs with many time steps.

The second graph in Figure 6 presents the impact of locality optimizations on a parallel version of MOLDYN. We found transformations are also very successful; improvements range from 30–160%. Results indicate locality is even more important for parallel programs, since slight differences in effectiveness of transformations yield large differences in parallel programs. GPART proved to be the best optimization for all applications when combining results from programs with both few and many time steps.

3.3 Cache-Conscious Memory Allocation

One of the advantages of pointer-based data structures is its locational transparency. The actual location of a node in memory is not important as long as all pointers to the node point to the location. As a result, data layout transformations have free rein to change data locations, as long as all pointers may be updated.

Algorithm Chilimbi *et al.* developed techniques to take advantage of this transparency with two run-time tools, CCMALLOC and CCMORPH [17]. CCMALLOC provides a memory allocation routine which tries to allocate a node close to a specified node, or to the beginning of a cache line if that is not possible. Since no pointers exist to unallocated data, the optimization is always safe. In comparison, CCMORPH targets high-level data structures such as linked lists and binary trees. It takes a pointer-based data structure and reorganizes elements by copying them to new locations and updating all internal pointers. Lists are linearized, and trees are clustered, with parents placed near children. In addition, coloring is used to avoid conflicts with top levels of a tree.

We believe cache-conscious structure layout is a good starting point for applying locality optimizations for pointer-based data structures. We developed and tested two extension to Chilimbi's allocation techniques. First, we are working on evaluating different heuristics for giving hints to CCMALLOC, the memory allocator. For instance, when tree leaves point to linked lists, the allocator may either keep the tree leaf with the head of the link list, or keep the linked list elements together.

Second, we extended CCMORPH to cluster directed acyclic graphs (DAGs) as well as trees. We use an auxiliary data structure to maintain a mapping of elements in the original DAG to elements in the new DAG. After reorganizing the original DAG, the algorithm makes a separate pass to fix the pointers in the new DAG.

Evaluation To evaluate different hints for CCMALLOC, we used the Olden benchmark MST, which computes the minimum spanning tree of a graph [10]. The graph is represented by a linked list of nodes, where each node contains a linked list representing the list of edges for that node (edge list). This code was used to evaluate giving different hints to the memory allocator. In the first scheme hints are given so that nodes in the list are placed together if possible. In the second scheme hints are given so that nodes and their edge lists are placed together if possible.

We measured cache misses for a graph of 200 nodes. Using *malloc*(), the L1 cache miss rate was 29%, and the L2 cache miss rate was 18%. Placing nodes close together with each other reduced the number of L1 misses by 4% and L2 misses by 20%, but increased the total number of references by 22%. In comparison, placing nodes close to their edge lists increased the number references similarly, but reduced the total number of L1 misses by 35% and L2 misses by 55%. Picking the right hints to pass to the allocator thus significantly affected the impact of locality optimizations.

Second, we evaluated the efficiency of our DAG optimizer. by we constructing a DAG benchmark similar to the tree benchmark used by Chilimbi [17] consisting of searches and scans for a randomly generated directed acyclic graph. The DAGs were constructed by first building a binary tree with some links missing (each node had a $1/2$ chance of having a single child), and then making a separate pass to point these unused links to nodes further down in the tree. An unused link was given a $1/3$ chance of becoming a DAG link. We evaluated three data layouts for nodes, random, depth-first, and clustered. Depth-first lays out nodes in order based on an inorder traversal of the DAG. Clustered layout places nodes near parents where possible.

When performing random searchs (from root to leaf) on a DAG with 100K nodes, L1 cache miss rates were 16.8%, 6.2%, and 4.7% for random, depth-first, and clustered data layouts, respectively. When traversing all nodes (inorder) of a DAG with 50K nodes, L1 cache miss rates were 20.4%, 5.3%, and 8.0% for random, depth-first, and clustered data layouts, respectively. We can draw two conclusions. First, locality optimizations are useful for DAGs. Second, taking traversal order into account can yield improved performance. For random traversals, clustering improves performance. But if traversals are known to be depth first (inorder), laying out nodes in the same fashion yield better performance.

4 Proposed Research Directions

We believe our preliminary results show software support is needed to exploit locality and improve cache performance in advanced scientific applications, but much work remains. We propose to explore research in two main areas, improving locality optimizations to better handle more complicated cases, and extending algorithms to support additional architecture features.

4.1 Improving Locality Optimizations

In general, we plan to more extensively test our locality optimizations by applying them to larger scientific programs. We are also working on increasing the quality of our algorithms for regular, irregular, and pointer-based computations.

4.1.1 Regular Computations

Our proposed research on regular computations extends our previous work on tiling and padding for iterative PDE solvers and linear algebra codes.

Tiling Algorithms Overall, our experiments show that tiling can improve performance for 3D iterative PDE solvers. We are working on extending tiling to new applications domains with regular memory access patterns, such as database scans/joins and image processing. We also plan to conduct a comprehensive comparison of copying versus padding as means of avoid cache conflicts.

Another area where tiling algorithms need to be improved is in automatically deriving cost models to evaluate tile dimensions. Our current cost models are based on estimated cross-interference between arrays [19, 71]. However, we can only calculate interference precisely for linear algebra codes. We would like to be able to come up with more precise measurements, possibly along the lines of cache miss equations [27].

In more complicated kernels, where multiple references bring data into cache, we need to partition the nonconflicting array tile between arrays. We believe this can be accomplished by reducing one tile dimension and then applying *inter-variable padding* so that each array accesses data mapping to its own partition in the array tile [70]. However, we need to evaluate the feasibility of this approach and incorporate it into our cost model.

Nonlinear Memory Layouts and Recursive Algorithms We also plan on comparing tiling with recently proposed schemes for nonlinear memory layouts [15], particularly coupled with recursive algorithms [90]. Standard array layout schemes such as column-major (Fortran) or row-major (C, C++) are very efficient, but may cause severe conflicts between higher dimensions for pathological array dimension sizes that are a multiple of the cache size. As a result, several researchers have proposed explicit nonlinear array layouts which avoid this problem. Similarly, researchers have also proposed recursive algorithms instead of traditional loop tiling in order to avoid these severe cache conflicts.

Both nonlinear memory layouts and recursive algorithms are usually justified by comparing against an unoptimized tiled code which does not apply either copying or padding. We intend to pursue a detailed experimental comparison against our best algorithms to determine whether they are indeed superior. In any case, we believe there are elements of the new algorithms which may be integrated with more traditional tiling to achieve improved performance. For instance, we can execute tiles in Morton order just as in nonlinear memory layouts [15] to exploit additional inter-tile reuse, but without paying the address calculation overhead of nontraditional array layouts.

4.1.2 Irregular Computations

Our proposed research on irregular computations extends our work on using graph partitioning algorithms to reorder data and computation for sparse meshes and n-body problems.

Space-Filling Curves Space-filling curves such as Morton ordering [15] have been shown to yield excellent performance when used to guide data and computation transformations for irregular scientific codes [58]. However, it is highly unlikely they can be applied automatically because they require geometric coordinate information for data points. As a result graph traversal and partitioning algorithms such as RCM and GPART are more practical locality optimizations for a compiler.

We plan, however, to integrate more features of space-filling curves in our locality optimizations. First, instead of lexicographically sorting loop iterations according to data access locations, we can rearrange computation using

Morton ordering. Unlike using space-filling curves for data, coordinate information is readily available since it is just the array indices accessed. Second, we also plan to develop simple user annotations which will inform the compiler that geometric coordination information is available, and where it may be found. Finally, we plan to develop algorithms to automatically choose granularities for space-filling curves which balance locality with overhead.

Sparse Matrix Linear Algebra Another avenue we intend to explore is to apply locality optimizations intended for sparse meshes and n-body codes to sparse linear algebra. Researchers have already shown that rows and columns of sparse matrices may be considered a graph, and successfully applied graph clustering algorithms to improve locality for sparse matrix codes [36]. We wish to determine whether the more powerful algorithms we have developed and evaluated are also suitable for sparse linear algebra codes.

Early experiments with sparse linear algebra show that tiling sparse matrix-vector multiplications with RCB can improve performance by 5–10% on an R10000, for instance. Correctness and performance issues are quite different than sparse mesh codes, however, because sparse matrix computations can rely on the underlying numerical properties of the matrix operations to reorder data and computation. It is also possible to "fill in" empty array elements with explicit zeros in order to create dense regular structures which may be solved efficiently, despite the additional space and computation engendered by the zeros. As a result much work remains to be done.

4.1.3 Pointer-Based Computations

Our proposed research on pointer-based computations attempt to extend the work of Chilimbi *et al.* in improving locality for C and C++ programs [17].

Compact Data Structures Standard memory allocators are designed for general use rather than high performance. As a result, they make design choices which are not desirable when attempt to exploit locality. For instance, when allocating nodes for a linked list using standard C *malloc*(), space is reserved for size information needed for the free list. In addition, memory allocated is rounded up to an even size. As a result, even when the allocator is called to allocate consecutive nodes in a linked list, the nodes are spaced farther apart than necessary, causing a loss of spatial locality. By using a specialized memory allocator which keeps track of memory size separately, nodes can be allocated much more closely, resulting in improved spatial locality.

In a preliminary experiment, we optimized the Olden benchmark HEALTH, which simulates the Colombian health care system [10]. A quad-tree (tree with 4 children) is used, with a linked list at each node in the tree. The original code uses *malloc*() to allocate the nodes of the linked list. Because of inefficiencies with the standard memory allocator, performance was poor even when nodes of a linked list are allocated consecutively. By recognizing nodes have fixed size, we can direct the memory allocator to allocate nodes efficiently, allow nodes to be packed more closely together.

With the original memory allocator, even after allocating nodes consecutively the L1 and L2 cache miss rates were 42% and 3.3%, respectively. In comparison, packing data using a special allocator reduced the L1 and L2 miss rates to 21% and 1.8%, respectively. It thus appears that memory allocators must be more intelligent in the size of data allocated, since it affects how closely data can be packed together. Identifying such opportunities using compiler analysis can be very profitable.

Exploit Traversal Order Instead of assuming all pointer-based structures are traversed in an unknown manner, determine whether information on the order tree nodes are accessed can improve performance. For instance, instead of always clustering all trees (parent with child), we may wish to cluster nodes with siblings if we know nodes are usually traversed breadth first. Alternatively, if nodes are traversed depth first along the leftmost child, we can cluster nodes with their leftmost children. If run-time profiling information finds tree traversals are along a given path, data can be reorganized to provide locality along that path. It may even be profitable to reorder both computation (loop iterations) and data layout in combination to achieve the best performance.

Better Compiler Allocation Hints Instead of just providing a single location (parent, predecessor) near where we would like new node allocated, we can also provide other locations such as siblings. This provides the compiler with more flexibility for data structures such as trees, particularly if nodes may have many children. In addition, instead of simply allocating a node at a new empty cache line when it cannot be placed near the desired location, we may direct the memory allocator to reserve larger amounts of space near the new node allocated. This is useful when we will be allocating nodes for a long linked list, or for trees with many children.

Existing allocation optimizations techniques apply to only linked lists and trees. We can also develop algorithms to provide better allocation patterns for DAGs, b-tree, quad-trees, and other pointer-based data structures to improve locality. For arbitrary connected graphs, graph partitioning algorithms may prove useful, just as in adaptive irregular Fortran codes.

Improve Compiler Analyses Current optimization techniques for pointer-based codes depend on user intervention to supply hints to the allocator, or to determine how and when data structures need to be reorganized. Integrating compiler analysis can help automate many of these decisions. In addition, many of the new optimizations proposed here can benefit from additional compiler analysis. For instance, compilers can try to determine whether data structures are modified during program execution or accessed from outside pointers, in order to establish whether lists/trees can be reorganized after allocation. Compilers can identify likely traversal order, type of data structures, and provide allocation hints. Much research is needed here to see how much can be done automatically in an advanced compiler.

Successfully deciding when locality optimizations should be applied is a difficult problem for many types of applications. We plan to also work on improving compile-time performance estimation of benefits of locality transformation, in order to help guide decisions on when they should be applied.

Support Adaptive Computations Finally, programs using pointer-based data structures frequently allocate and deallocate data during the course of the computation. How frequently locality optimizations such as tree clustering should be reapplied is a difficult issue. We plan to evaluate adapting cost models and feedback control systems developed for adaptive irregular codes [32] to guide locality optimizations for dynamic pointer-based codes. The problems are very similar, particularly the key point that locality optimizations are optional, and are affected by the rate of change in data structures.

4.2 Additional Architecture Features

In addition working on improvements to our locality optimizations for a variety of advanced scientific codes, we also wish to extend our algorithms to handle a number of important architecture features.

4.2.1 Software Prefetching

Prefetching attempts to hide the latency of memory accesses by anticipating what data is needed and preloading it into the cache ahead of time. Memory stalls are thus overlapped with computation and other memory accesses. Hardware prefetching relies on long cache lines to capture spatial locality, and works well for contiguous memory accesses. Software prefetching relies on the programmer or compiler to insert explicit prefetch instructions into the code. It has been shown to be effective in reducing memory stall cycles for both sequential and parallel applications, particularly for scientific programs making regular memory accesses [9, 62, 61]. Techniques have also been developed to apply prefetching to pointer-based data structures [54].

As support for software prefetching becomes more common, we wish to determine the usefulness of software prefetching combined with our locality optimizations, and determine how they interact. Working with Prof. Don Yeung, we have some preliminary results, using a simulator based on SimpleScalar [7]. We simulate a 1 Ghz processor

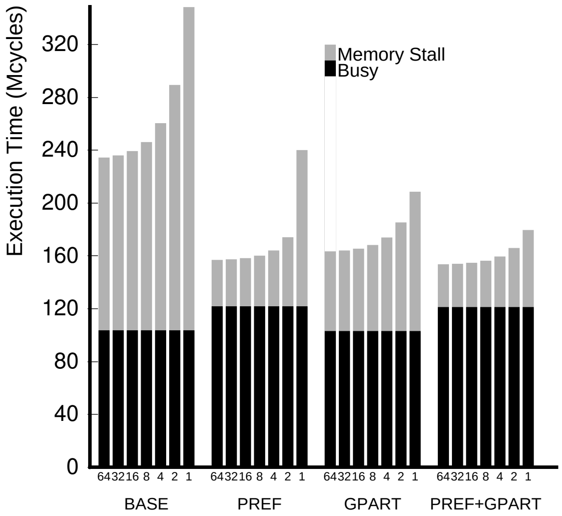


Figure 7 Execution times for MOLDYN

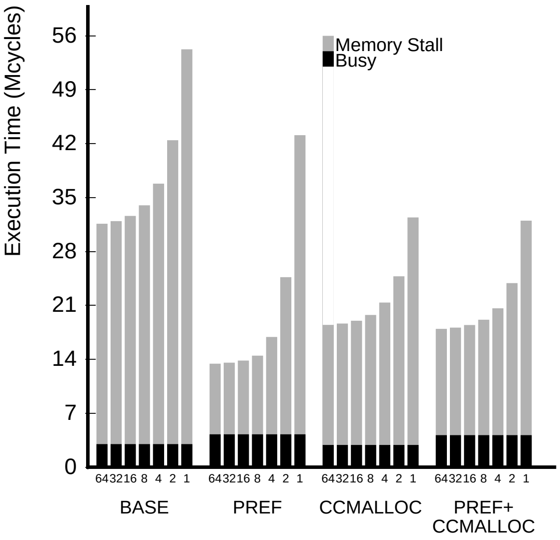


Figure 8 Execution times for HEALTH

and memory bandwidths ranging from 1–64 GB/sec. Results are shown in Figures 7 and 8 for an irregular code (MOLDYN) and pointer-based code (HEALTH).

Execution time is plotted along the y-axis, with execution time broken down into memory stall and actual computation. Program version and available bandwidth (from 64 GB/sec to 1 GB/sec) is plotted along the x-axis. Different versions of each program represent versions with software prefetching and locality optimizations individually and together. MOLDYN is optimized using an inspector applying the GPART reordering algorithm. The Olden benchmark HEALTH is optimized by compacting list data using allocation hints.

As the results show, performance generally improves as available memory bandwidth increases. When available bandwidth is low, locality optimizations achieve the best performance. When bandwidth is high, prefetching obtains the best performance. Prefetching versions typically have higher "BUSY" times because of the added overhead for issuing prefetch instructions. Since memory bandwidth will probably continue to drop relative to processor speed, locality optimizations should become more important. However, we need to design locality optimizations that successfully combine with software prefetching to achieve the best performance. For instance, tile size selection can affect software prefetching overhead, so larger tiles which have lower overhead are preferred. We intend to study these interactions with more comprehensive experiments.

4.2.2 Larger Set Associativity, Multi-Level Caches, TLBs

As memory speeds continue to lag relative to processor speed, computer designers are putting more effort in hardware features to combat memory latency. We would like to extend our locality optimizations to take these hardware features into account. First, faced with a choice between larger caches or higher set associativity, some microprocessor designers are beginning to opt for higher set associativity to reduce the chance of severe cache conflicts. While L1 caches remain direct-mapped, the DEC Alpha 21164 has a 3-way associative L2 cache, and the Intel Itanium processor ahs a 6-way L2 cache. Most current locality algorithms treat caches as direct-mapped if they consider set associativity at all. While treating caches as direct-mapped will not cause any misses, it should be possible to exploit the higher associativity. We are working to develop such new algorithms, particularly for avoiding inter-array conflicts.

Another feature of advanced microprocessors is the introduction of multiple levels of cache. In an earlier study we determined that most of the benefits of cache optimizations can be achieved by simply targeting the smallest level of cache where data reuse can be obtained [72]. However, an important exception to this rule is trying to improve translation look-aside buffer (TLB) performance as well as cache performance [60]. Because L2 and L3 caches are becoming so large, it is entirely possible to run out of TLB entries before filling cache if data accesses are widely scattered. On some processors TLBs are only 4-way instead of fully associative, increasing chances of TLB misses significantly. We intend to examine this issue and work on developing tiling and padding algorithms which can optimize both cache and TLB performance simultaneously for the best overall result.

4.2.3 Parallel and Cluster Architectures

Finally, locality optimizations must also take into account parallelism and the complex memory hierarchies of parallel and cluster architectures, where nonlocal memory and interprocessor communication provides another orthogonal issue to consider besides deep memory hierarchies. Pointer-based data locality optimizations on parallel codes is an unexplored area. In general, we believe locality optimizations should benefit parallel codes, since improving locality of reference can actually also reduce interprocessor communication [30, 47]. However, locality optimizations must ensure they do not detract from the ability to exploit parallelism.

Another issue which may impact locality optimizations is the presence of false sharing, which can cause ping-pong misses [38]. Locality optimizations need to be careful not to create false sharing, and especially target eliminating false sharing misses where possible. Finally, because the cost of nonlocal memory accesses is much higher, more expensive locality optimizations may prove profitable. This proved to be the case for locality optimizations for irregular mesh codes, where high-overhead partitioning algorithms such as RCB were much more competitive with GPART on a DEC Alpha multiprocessor than on a uniprocessor.

5 Outreach

To establish the effectiveness our techniques, we plan to target programs of increasing size and complexity. We will begin with kernels computations (e.g., multigrid, molecular dynamics, OLDEN pointer suite), move on to the SPEC2000 benchmark suite (both integer and floating point), and finally target commonly used mathematical libraries and application programs.

We are implementing our optimizations in COSMIC, an optimizing compiler based on the Stanford SUIF parallelizing compiler [86]. To make the results of our research more widely accessible, we plan on releasing our optimizations as new SUIF compiler passes. Since SUIF is part of the National Compiler Infrastructure project, passes in SUIF will be usable by many compiler research groups.

We also plan on releasing the code for our run-time data and computation transformations as a run-time library, with clearly documented interfaces. Our library will contain optimizations for both irregular meshes and pointer-based codes, and will be usable by either compilation systems or programmers who wish to tune their application performance by hand. We will model our software distribution along the lines used by Prof. Joel Saltz in distributing his CHAOS run-time library [21].

Finally, we intend to collaborate with Prof. Jeff Hollingsworth, who is working on user performance tools to isolate and display memory bottlenecks using hardware cache miss counters [6]. Similar tools such as MemSpy have proven very useful in helping users identify problematic data layouts and computation patterns with poor cache performance [56]. We intend to both use their tools to identify optimization targets, and integrate our analyses with their tool to provide advice to the user on possible causes and fixes for their memory bottlenecks.

Our goal of providing software support for locality is worth pursuing because it holds the promise of making high-performance computing more accessible for scientists and engineers. Because of trends in computer architectures, lessons learned here are also likely to prove very useful for other application domains, including image processing and high-performance databases.

6 Related Work

There has been much work on improving locality in scientific applications [3, 24, 25, 26, 39, 40, 57, 67, 69, 70, 79, 87]. Here we will focus on the work which is most relevant to our proposed research.

A number of researchers have investigated tiling as a means of exploiting reuse. Lam, Rothberg, Wolf show conflict misses can severely degrade the performance of tiling [51]. Wolf and Lam analyze temporal and spatial reuse, and apply tiling when necessary to capture outer loop reuse [87], Coleman and McKinley select rectangular non-conflicting tile sizes [19] while others focus on using a portion of cache [88]. Temam *et al.* analyze the program to determine whether a tile should be copied to a contiguous buffer [80].

Kodukula *et al.* present a technique called data shackling which is very effective at generating tiled code for complex linear algebra codes [48]. Sarkar describes data locality optimizations used in the IBM XL Fortran compilers, including loop transformations and tiling [74]. Mitchell *et al.* discussed the interactions of multi-level tiling for several goals, such as cache, TLB, and parallelism [60].

Song and Li extending tiling techniques to handle multiple loop nests [76]. In many cases their technique can exploit reuse across multiple iterations of the time-step loop, yielding major improvements. Their technique does not currently apply to 3D arrays, and they concentrate on only L2 cache since L1 cache is too small to provided reuse for their necessarily large tiles. We intend to investigate combining our tiling techniques with theirs. Weiß *et al.* explored program transformations for improving the performance numerical algorithms on hierarchical memories [85]. They applied transformations by hand based on domain-specific applications knowledge. We attempt to develop similar techniques which can be automated in a compiler.

Compared to existing research, we attempt to find compiler and run-time solutions to locality problems specific to 3D iterative solvers. We point out several difficult problems and propose solutions.

Researchers have investigated both run-time [21, 35] and compiler [14, 33, 53, 52, 68] support for irregular computations, mostly in the context of message-passing multiprocessors. Researchers have also examine efficiently supporting irregular computations on software DSMs [5, 30, 53, 63], including hybrid regular-irregular codes [12, 50, 78]. Domain decomposition algorithms such as RCB and METIS have been used by researchers for load balancing, reducing interprocessor communication, and many other uses [42, 66, 65, 75]. A few researchers have investigated improving locality for irregular scientific applications. Das *et al.* investigated data/computation reordering for unstructured euler solvers [20], using RCM and lexicographical sort. Al-Furaih and Ranka studied using METIS and breadth-first-search to partition irregular computations and improve locality [1].

Ding and Kennedy explored applying dynamic copying (packing) of data elements based on loop traversal order [22]. They also developed algorithms for reorganizing single arrays into multi-dimensional arrays depending on their access patterns [23]. Mellor-Crummey *et al.* use space-filling curves to map multidimensional data to memory [58]. However, space-filling curves must be applied by hand using geometric coordinate information. Mitchell *et al.* improved locality using bucket sorting to reorder loop iterations in irregular computations containing a single irregular access per loop iteration [59]. Nicol and Saltz investigated algorithms to remap data and computation for adaptive parallel codes to reduce load imbalance[64]. We use a similar approach for our on-the-fly optimization technique, but use it for locality instead of load imbalance.

Compared to existing research on locality optimizations for irregular codes, we propose a suite of algorithms which differ in precision and overhead, including new algorithms designed for low overhead and high locality. These algorithms are vital for adaptive codes, where low overhead is crucial. We test effects on meshes from real applications for a number of computation kernels.

Few researchers have investigated data layout transformations for pointer-based data structures. Chilimbi, Hill, and Larus investigate allocation-time and run-time techniques to improve locality for linked lists and trees [17]. We propose extension to their work. Calder *et al.* use profiling to guide layout of global and stack variables to avoid conflicts [8]. Carlisle *et al.* investigate parallel performance for pointer-based codes in Olden [10].

7 Conclusions

In this paper, we have examined locality optimizations needed for three features of advanced scientific applications (3D arrays, irregular accesses, and pointers). Preliminary experimental evaluation is very encouraging, but much work remains to automate and improve these compiler and run-time systems. Our goal of providing software support for locality is worth pursuing because it holds the promise of making high-performance computing more accessible for scientists and engineers. Because of trends in computer architectures, lessons learned here are also likely to prove very useful for other application domains, including image processing and high-performance databases.

References

[1] I. Al-Furaih and S. Ranka. Memory hierarchy management for iterative graph structures. In *Proceedings of the 12th International Parallel Processing Symposium*, Orlando, FL, April 1998.

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

[3] D. Bacon, J.-H. Chow, D.-C. Ju, K. Muthukumar, and V. Sarkar. A compiler framework for restructuring data declarations to enhance cache and TLB effectiveness. In *Proceedings of CASCON'94*, Toronto, Canada, October 1994.

[4] T. Bonk and U. Rüde. Performance analysis and optimization of numerically intensive programs. SFB Bericht 342/26/92 A, Institut für Informatik, TU München, November 1992.

[5] P. Brezany, M. Gerndt, V. Sipkova, and H. Zima. SUPERB support for irregular scientific computations. In *Proceedings of the 1992 Scalable High Performance Computing Conference*, Williamsburg, VA, April 1992.

[6] B. Buck and J. Hollingsworth. Using hardware performance monitors to isolate memory bottlenecks. In *Proceedings of SC'00*, Dallas, TX, November 2000.

[7] Doug Burger and Todd M. Austin. The SimpleScalar Tool Set, Version 2.0. CS TR 1342, University of Wisconsin-Madison, June 1997.

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

[9] D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, Santa Clara, CA, April 1991.

[10] M. Carlisle, A. Rogers, J. Reppy, and L. Hendren. Early experiences with Olden. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

[11] S. Carr and K. Kennedy. Compiler blockability of numerical algorithms. In *Proceedings of Supercomputing '92*, Minneapolis, MN, November 1992.

[12] D. Chakrabarti, N. Shenoy, A. Choudhary, and P. Banerjee. An efficient uniform run-time scheme for mixed regular-irregular applications. In *Proceedings of the 1998 ACM International Conference on Supercomputing*, Melbourne, Australia, July 1998.

[13] R. Chandra, D.-K. Chen, R. Cox, E. Maydan, N. Nedeljkovic, and J. Anderson. Data distribution support on distributed shared memory multiprocessors. In *Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation*, Las Vegas, NV, June 1997.

[14] S. Chandra and J.R. Larus. Optimizing communication in HPF programs for fine-grain distributed shared memory. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Las Vegas, NV, June 1997.

[15] S. Chatterjee, V. Jain, A. Lebeck, S. Mundhra, and M. Thottethodi. Nonlinear array layouts for hierarchical memory systems. In *Proceedings of the 1999 ACM International Conference on Supercomputing*, Rhodes, Greece, June 1999.

[16] T. Chilimbi, B. Davidson, and J. Larus. Cache-conscious structure definition. In *Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation*, Atlanta, GA, May 1999.

[17] T. Chilimbi, M. Hill, and J. Larus. Cache-conscious structure layout. In *Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation*, Atlanta, GA, May 1999.

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

[19] S. Coleman and K. S. McKinley. Tile size selection using cache organization and data layout. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, La Jolla, CA, June 1995.

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

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

[22] C. Ding and K. Kennedy. Improving cache performance of dynamic applications with computation and data layout transformations. In *Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation*, Atlanta, GA, May 1999.

[23] C. Ding and K. Kennedy. Inter-array data regrouping. In *Proceedings of the Twelfth Workshop on Languages and Compilers for Parallel Computing*, San Diego, CA, August 1999.

[24] J. Ferrante, V. Sarkar, and W. Thrash. On estimating and enhancing cache effectiveness. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing, Fourth International Workshop*, Santa Clara, CA, August 1991. Springer-Verlag.

[25] 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(5):587–616, October 1988.

[26] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: An analytical representation of cache misses. In *Proceedings of the 1997 ACM International Conference on Supercomputing*, Vienna, Austria, July 1997.

[27] S. Ghosh, M. Martonosi, and S. Malik. Precise miss analysis for program transformations with caches of arbitrary associativity. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, San Jose, CA, October 1998.

[28] M. Hall, J. Anderson, S. Amarasinghe, B. Murphy, S.-W. Liao, E. Bugnion, and M. Lam. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, 29(12), December 1996.

[29] H. Han and C.-W. Tseng. Compile-time synchronization optimizations for software DSMs. In *Proceedings of the 12th International Parallel Processing Symposium*, Orlando, FL, April 1998.

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

[31] H. Han and C.-W. Tseng. A comparison of locality transformations for irregular codes. In *Proceedings of the 5th Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, Rochester, NY, May 2000.

[32] H. Han and C.-W. Tseng. Improving locality for adaptive irregular codes. In *Proceedings of the Thirteenth*

Workshop on Languages and Compilers for Parallel Computing, White Plains, NY, August 2000.

[33] R. v. Hanxleden. Handling irregular problems with Fortran D — A preliminary report. In *Proceedings of the Fourth Workshop on Compilers for Parallel Computers*, Delft, The Netherlands, December 1993.

[34] Y. Hu, S. L. Johnsson, and S.-H. Teng. High Performance Fortran for highly irregular problems. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Las Vegas, NV, June 1997.

[35] Y.-S. Hwang, B. Moon, S. Sharma, R. Ponnusamy, R. Das, and J. Saltz. Runtime and language support for compiling adaptive irregular programs on distributed memory machines. *Software—Practice and Experience*, 25(6):597–621, June 1995.

[36] E. Im and K. Yelick. Model-based memory hierarchy optimizations for sparse matrices. In *Proceedings of the 1998 Workshop on Profile and Feedback-Directed Compilation*, Paris, France, October 1998.

[37] F. Irigoin and R. Triolet. Supernode partitioning. In *Proceedings of the Fifteenth Annual ACM Symposium on the Principles of Programming Languages*, San Diego, CA, January 1988.

[38] T. Jeremiassen and S. Eggers. Reducing false sharing on shared memory multiprocessors through compile time data transformations. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, CA, July 1995.

[39] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. Improving locality using loop and data transformations in an integrated framework. In *Proceedings of the 31th IEEE/ACM International Symposium on Microarchitecture*, Dallas, TX, November 1998.

[40] M. Kandemir, J. Ramanujam, and A. Choudhary. A compiler algorithm for optimizing locality in loop nests. In *Proceedings of the 1997 ACM International Conference on Supercomputing*, Vienna, Austria, July 1997.

[41] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. In *Proceedings of the 24th International Conference on Parallel Processing*, Oconomowoc, WI, August 1995.

[42] G. Karypis and V. Kumar. Multi-level k-way hypergraph partitioning. In *Proceedings of SC'98*, Orlando, FL, November 1998.

[43] P. Keleher. The relative importance of concurrent writers and weak consistency models. In *16th International Conference on Distributed Computing Systems*, Hong Kong, May 1996.

[44] P. Keleher. Update protocols and iterative scientific applications. In *Proceedings of the 12th International Parallel Processing Symposium*, Orlando, FL, April 1998.

[45] P. Keleher and C.-W. Tseng. Enhancing software DSM for compiler-parallelized applications. In *Proceedings of the 11th International Parallel Processing Symposium*, Geneva, Switzerland, April 1997.

[46] K. Kennedy and U. Kremer. Automatic data layout for High Performance Fortran. In *Proceedings of Supercomputing '95*, San Diego, CA, November 1995.

[47] K. Kennedy and K. S. McKinley. Optimizing for parallelism and data locality. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, Washington, DC, July 1992.

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

[49] I. Kodukula, K. Pingali, R. Cox, and D. Maydan. An experimental evaluation of tiling and shacking for memory hierarchy management. In *Proceedings of the 1999 ACM International Conference on Supercomputing*, Rhodes, Greece, June 1999.

[50] A. Lain and P. Banerjee. Techniques to overlap computation and communication in irregular iterative applications. In *Proceedings of the 1994 ACM International Conference on Supercomputing*, Manchester, UK, July 1994.

[51] M. Lam, E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, Santa Clara, CA, April 1991.

[52] Y. Lin and D. Padua. Compiler analysis of irregular memory accesses. In *Proceedings of the SIGPLAN '00*

Conference on Programming Language Design and Implementation, Vancouver, Canada, June 2000.

[53] H. Lu, A. Cox, S. Dwarkadas, R. Rajamony, and W. Zwaenepoel. Compiler and software distributed shared memory support for irregular applications. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Las Vegas, NV, June 1997.

[54] C.-K. Luk and T. Mowry. Compiler-based prefetching for recursive data structures. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, Boston, MA, October 1996.

[55] N. Manjikian and T. Abdelrahman. Fusion of loops for parallelism and locality. *IEEE Transactions on Parallel and Distributed Systems*, 8(2):193–209, February 1997.

[56] M. Martonosi, A. Gupta, and T. Anderson. MemSpy: Analyzing memory system bottlenecks in programs. In *Proceedings of the 1997 ACM SIGMETRICS Conference on Measurement & Modeling Computer Systems*, Newport, RI, June 1992.

[57] K. S. McKinley, S. Carr, and C.-W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.

[58] J. Mellor-Crummey, D. Whalley, and K. Kennedy. Improving memory hierarchy performance for irregular applications. In *Proceedings of the 1999 ACM International Conference on Supercomputing*, Rhodes, Greece, June 1999.

[59] N. Mitchell, L. Carter, and J. Ferrante. Localizing non-affine array references. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, Newport Beach , LA, October 1999.

[60] N. Mitchell, L. Carter, J. Ferrante, and K. Högstedt. Quantifying the multi-level nature of tiling interactions. In *Proceedings of the Tenth Workshop on Languages and Compilers for Parallel Computing*, Minneapolis, MN, August 1997.

[61] T. Mowry. Tolerating latency in multiprocessors through compiler-inserted prefetching. *IEEE Transactions on Computer Systems*, 16(1):55–92, February 1998.

[62] T. Mowry and A. Gupta. Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 12(2):87–106, June 1991.

[63] S. Mukherjee, S. Sharma, M. Hill, J. Larus, A. Rogers, and J. Saltz. Efficient support for irregular applications on distributed-memory machines. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, CA, July 1995.

[64] D. Nicol and J. Saltz. Dynamic remapping of parallel computations with varying resource demands. *IEEE Transactions on Computers*, 37(9):1073–1087, September 1988.

[65] C. Ou et al. Fast and parallel mapping algorithms for irregular and adaptive problems. In *Journal of Supercomputing*, 1994.

[66] C. Ou, M. Gunwani, and S. Ranka. Architecture independent locality-improving transformations of computational graphs embedded in k-dimensions. In *Proceedings of the 1995 ACM International Conference on Supercomputing*, Barcelona, Spain, July 1995.

[67] R. Panda, H. Nakamura, N. Dutt, and A. Nicolau. Augmenting loop tiling with data alignment for improved cache performance. *IEEE Transactions on Computers*, 48(2):142–149, February 1999.

[68] M. Rinard and P. Diniz. Commutativity analysis: A new analysis technique for parallelizing compilers. *ACM Transactions on Programming Languages and Systems*, 19(6):942–992, November 1997.

[69] G. Rivera and C.-W. Tseng. Data transformations for eliminating conflict misses. In *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, Montreal, Canada, June 1998.

[70] G. Rivera and C.-W. Tseng. Eliminating conflict misses for high performance architectures. In *Proceedings of the 1998 ACM International Conference on Supercomputing*, Melbourne, Australia, July 1998.

[71] G. Rivera and C.-W. Tseng. A comparison of compiler tiling algorithms. In *Proceedings of the 8th International Conference on Compiler Construction (CC'99)*, Amsterdam, The Netherlands, March 1999.

[72] G. Rivera and C.-W. Tseng. Locality optimizations for multi-level caches. In *Proceedings of SC'99*, Portland, OR, November 1999.

[73] G. Rivera and C.-W. Tseng. Tiling optimizations for 3d scientific computations. In *Proceedings of SC'00*, Dallas, TX, November 2000.

[74] V. Sarkar. Automatic selection of higher order transformations in the IBM XL Fortran compilers. *IBM Journal of Research and Development*, 41(3):233–264, May 1997.

[75] J. P. Singh, C. Holt, T. Totsuka, A. Gupta, and J. Hennessy. Load balancing and data locality in adaptive hierarchical n-body methods: Barnes-hut, fast multipole, and radiosity. *Journal of Parallel and Distributed Computing*, June 1995.

[76] Y. Song and Z. Li. New tiling techniques to improve cache temporal locality. In *Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation*, Atlanta, GA, May 1999.

[77] L. Stals, U. Rüde, C. Weiss, and H. Hellwagner. Data local iterative methods for the efficient solution of partial differential equations. In *Proceedings of the Eighth Biennial Computational Techniques and Applications Conference*, Adelaide, Australia, September 1997.

[78] E. Su, A. Lain, S. Ramaswamy, D. J. Palermo, E. W. Hodges IV, and P. Banerjee. Advanced compilation techniques in the PARADIGM compiler for distributed-memory multicomputers. In *Proceedings of the 1995 ACM International Conference on Supercomputing*, Barcelona, Spain, July 1995.

[79] O. Temam, C. Fricker, and W. Jalby. Cache interference phenomena. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement & Modeling Computer Systems*, Santa Clara, CA, May 1994.

[80] O. Temam, E. Granston, and W. Jalby. To copy or not to copy: A compile-time technique for assessing when data copying should be used to eliminate cache conflicts. In *Proceedings of Supercomputing '93*, Portland, OR, November 1993.

[81] E. Torrie, C.-W. Tseng, M. Martonosi, and M. Hall. Evaluating the impact of advanced memory systems on compiler-parallelized codes. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, Limassol, Cyprus, June 1995.

[82] C.-W. Tseng. Compiler optimizations for eliminating barrier synchronization. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, CA, July 1995.

[83] C.-W. Tseng. Efficient machine-independent programming of high-performance multiprocessors. Technical Report CS-TR-3545, Dept. of Computer Science, University of Maryland at College Park, October 1995.

[84] C.-W. Tseng, J. Anderson, S. Amarasinghe, and M. Lam. Unified compilation techniques for shared and distributed address space machines. In *Proceedings of the 1995 ACM International Conference on Supercomputing*, Barcelona, Spain, July 1995.

[85] C. Weiß, W. Karl, M. Kowarschik, and U. Rüde. Memory characteristics of iterative methods. In *Proceedings of SC'99*, Portland, OR, November 1999.

[86] R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S.-W. Liao, C.-W. Tseng, M. Hall, M. Lam, and J. Hennessy. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 29(12):31–37, December 1994.

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

[88] M. E. Wolf, D. Maydan, and D.-K. Chen. Combining loop transformations considering caches and scheduling. In *Proceedings of the 29th IEEE/ACM International Symposium on Microarchitecture*, Paris, France, December 1996.

[89] M. J. Wolfe. More iteration space tiling. In *Proceedings of Supercomputing '89*, Reno, NV, November 1989.

[90] Q. Yi, V. Adve, and K. Kennedy. Transforming loops to recursion for multi-level memory hierarchies. In *Proceedings of the SIGPLAN '00 Conference on Programming Language Design and Implementation*, Vancouver, Canada, June 2000.