Improving Fine-Grained Irregular Shared-Memory Benchmarks by Data Reordering*

Y. Charlie Hu, Alan Cox and Willy Zwaenepoel

Department of Computer Science
Rice University
Houston, Texas 77005
{ychu, alc, willy}@cs.rice.edu

Abstract

We demonstrate that data reordering can substantially improve the performance of fine-grained irregular shared-memory benchmarks, on both hardware and software shared-memory systems. In particular, we evaluate two distinct data reordering techniques that seek to co-locate in memory objects that are in close proximity in the physical system modeled by the computation. The effects of these techniques are increased spatial locality and reduced false sharing.

We evaluate the effectiveness of the data reordering techniques on a set of five irregular applications from SPLASH-2 and Chaos. We implement both techniques in a small library, allowing us to enable them in an application by adding less than 10 lines of code. Our results on one hardware and two software shared-memory systems show that, with data reordering during initialization, the performance of these applications is improved by 12%–99% on the Origin 2000, 30%–366% on TreadMarks, and 14%–269% on HLRC.

1. Introduction

Over the last few years we have seen the development of several benchmark suites for shared-memory parallel systems [25, 26, 30, 33]. These benchmark suites have proven to be invaluable research tools, providing a basis for comparison between various shared-memory architectures (e.g., [13, 25]). Results from these benchmark suites have, however, also been taken as a measure of the performance that can be obtained on shared-memory architectures for the classes of applications that these benchmark programs represent.

This paper demonstrates that, for the class of fine-grained irregular applications, the current set of benchmark programs under-estimates the performance of shared-memory architectures, both hardware and software, and that major improvements can be obtained by using simple *data reordering* techniques. These data reordering techniques relocate data in memory in order to improve spatial locality and reduce false sharing. The techniques apply to a large class of applications, making it attractive to encapsulate them in a library, thereby reducing the modifications to the actual benchmark programs to a few lines of code. Furthermore, they are platform-independent and provide benefits both on hardware and software shared-memory platforms. The precise choice of which data reordering technique to choose for best results depends on some of the characteristics of the application and the platform. Determining these characteristics is, however, straightforward, and therefore the programmer can easily make the right choice.

We have incorporated a data reordering library in a number of fine-grained irregular programs from these standard benchmark suites. In particular, we have used Barnes-Hut, FMM, and Water-Spatial from SPLASH-2 [33], and Moldyn and Unstructured from the Chaos benchmark suite [11]. We have evaluated the modified programs on a hardware shared-memory machine (a 16-processor SGI Origin 2000 [23]) and two software shared-memory systems (TreadMarks [1] and HLRC [35] on a cluster of 16 Pentium II-based computers). Our results show that data ordering during initialization improves the performance of these applications by 12% – 99% on the Origin 2000, by 30% – 366% under TreadMarks, and by 14% - 269% under HLRC. These improvements result from better spatial locality and reduced false sharing, both brought about by data reordering. These benefits far outweigh the cost of executing the reordering code.

This paper makes contributions in terms of benchmarking, providing improved benchmarks for shared-memory parallel systems. The new benchmarks and data reordering library will be made available, and should prove more accurate in predicting the performance of shared-memory architectures for fine-grained irregular applications. The paper also makes contributions in the area of optimization techniques for shared-memory parallel programming, showing that data reordering techniques can be applied with substantial benefit to fine-grained irregular shared memory programs, and presenting guidelines for the choice of the appropriate reordering technique.

The rest of this paper is organized as follows. Section 2 describes the class of fine-grained irregular applications that we are concerned with in this paper and the problems occurring with the standard benchmark programs on shared-memory architectures. Section 3 describes the data reordering techniques that we use. Section 4 describes the experimental environment. We present the results of our experiments in Section 5. We discuss related work in Section 6 and conclude in Section 7.

2. Fine-grained Irregular Applications

The applications considered in this paper solve some computational problemsq in some given physical domains. The laws of physics cause these applications to have good locality in *physical* space: the objects interact mostly with other objects close to them in physical space. For instance, the gravitational force quickly decays with the distance, and approximations using only short-range interactions produce accurate results. Such applications are typically parallelized in one of two ways.

The first approach, which we refer to as Category 1, tries to partition the computation between the processors in an intelligent way, in particular, in such a way that physically close objects are handled by the same processor. This approach reduces the number of interactions between the processors, but requires a separate data structure to maintain the physical proximity relationships, in addition to the object array that records the relevant characteristics of each object. This additional data structure typically consists of a collection of *cells*, each one corresponding to a physically contiguous region, and maintained either in a tree or in a grid. A tree is used in Barnes-Hut and FMM; a grid is used in Water-Spatial.

The second approach does not make an attempt at sophisticated computation partitioning. Instead, it typically uses a simple block partitioning of the object array. The object array entry for a specific object maintains a list of objects in physical proximity, with which interactions are computed. This computational structure occurs in the moldyn and the unstructured benchmarks from the Chaos benchmark set.

For both categories of programs, data reordering, i.e., reordering the locations in memory of the objects in the object array, achieves substantial benefits. Although for both categories the benefits arise from better memory locality, more detailed inspection reveals that the nature of this improvement in spatial locality is different for both categories. Therefore, different reordering methods achieve the best results for each category.

2.1. Category 1: Sophisticated Computation Partition

The first category covers hierarchical N-body algorithms such as the Barnes-Hut method and the Fast Multipole Method [16] and grid-based applications such as Water-Spatial. In these applications, the computation is partitioned using an additional data structure, such as the tree in hierarchical N-body algorithms or the grid in grid-based N-body algorithms, as opposed to directly partitioned on the object array. The computation is partitioned to have each processor work on some subtrees or subgrids corresponding to some physically contiguous domain. Since objects interact with physically close objects, each processor updates particles or mesh points located within and near the boundary of that physically contiguous domain. The objects are, however, often initialized and stored in the shared address space in some random order, and physically adjacent particles are scattered all through memory.

The combination of random placement in memory and computation partitioning grouping physically adjacent bodies has two harmful effects. First, the bodies updated by a particular processor are spread throughout memory, giving rise to serious false sharing, especially in page-based software shared memory systems. False sharing occurs when two or more processors access different locations in the same consistency unit, and at least one of these accesses is a write. It is particularly serious in page-based software shared memory systems, because the consistency unit, a page, contains many bodies, but it is also present on hardware shared memory systems where a cache line may contain multiple bodies. Second, in order to update its own objects, a processor needs to read its (physically) neighboring objects. These neighboring objects are spread over many different consistency units, which gives rise to a large number of access misses. This effect is more pronounced on systems with a small consistency unit.

We use the Barnes-Hut benchmark to illustrate the problem on a 4-processor software shared memory system. Barnes-Hut [29] is an N-body simulation code using the hierarchical Barnes-Hut method [2]. A shared tree data structure is used to represent the recursively decomposed subdomains (cells) of the three-dimensional physical domain containing all the particles. The other shared data structure is an array of particles corresponding to the leaves of the

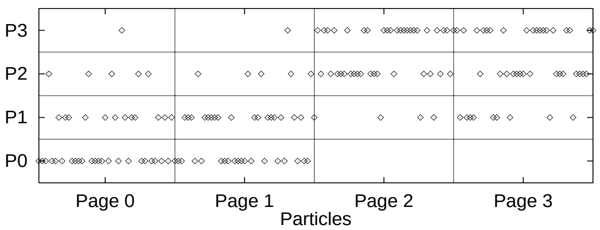

Figure 1: Locations of the 168 particles in each of the four 4-kilobyte pages to be updated by the four processors. We assume particles do not cross page boundary, and a page contains 42 96-byte particles.

tree. Each iteration is divided into two steps with barrier synchronization after each.

1. Build tree: a single processor reads all of the particles and rebuilds the tree.[1]

2. Force evaluation: the processors first divide up the particles in parallel through an in-order traversal of the tree. Specifically, the ith processor assigns itself the ith subset of contiguous subtrees weighted according to the workload recorded from the previous iteration. The force evaluation for each of the particles then involves some partial traversal of the tree. In this step each processor reads a large portion of the tree.

The input particles are often generated and stored in the shared particle array in random order, unrelated to their physical location.

The Barnes-Hut method is typically used in astrophysics to simulate the evolution of galaxies. Therefore, we focus on the three-dimensional two-Plummer distribution test case of the benchmark here. A single Plummer particle distribution is used to model a single galaxy of stars where the density of stars grows exponentially in moving towards the center of the galaxy. This example consists of 168 stars, each represented by a 96-byte record in the object array, occupying in total 4 4Kbyte pages. The actual generation of the coordinates of the particles uses a random number generator. Figure 1 shows the locations in memory to be updated by each processor during the second iteration: each processor needs to update locations in several pages.

In another example, we ran the code with 32768 bodies which collectively occupy 384 8Kbyte virtual memory pages. We then plot the number of processors sharing each of the 384 pages when running on 2, 4, 8, and 16 processors, respectively. The plot, shown in Figure 2, shows that each page is falsely shared by more than half of processors!

[1]Sequential tree building is commonly used for software shared memory, although this differs from the original benchmark code, which builds the tree in parallel.

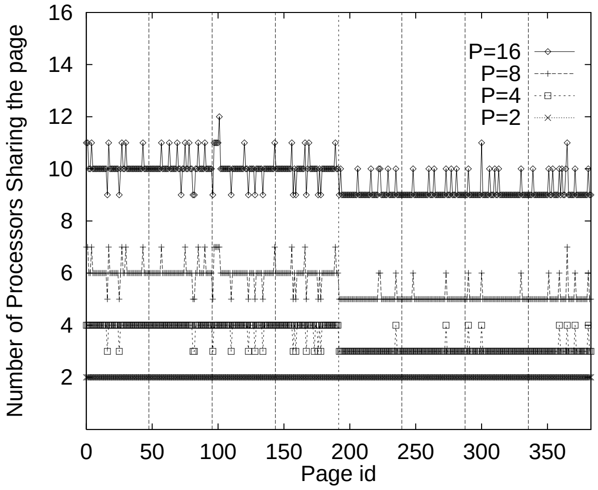

Figure 2: Number of processors sharing each of the 384 pages for the particle array of Barnes-Hut with 32K particles on 2, 4, 8, and 16 processors.

2.2. Category 2: Block Computation Partition

The second category covers irregular applications that simply (block-)partition the object array among the processors. Examples of this category include short-range N-body simulation codes that directly operate on the particles, such as molecules in Moldyn and unstructured, and unstructured mesh applications typical in computational fluid dynamics, which directly operate on the mesh points. In the actual implementation, due to the short-range nature of the applications, there is often an auxiliary data structure called the *interaction list* that is pre-constructed to contain indices pointing to spatially adjacent data units, and the data reference of the actual computation is often via this interaction list.

Similar to applications in Category 1, the data units in the main data structure are typically initialized in some random order that does not preserve the physical locality of the points they represent in space. With a block partitioning of the object array, the processors exhibit good locality in terms of writing to memory (they just write to elements in their block), but very poor locality in terms of reading, because the particle array entries that represent physically near objects are spread throughout the array. The result is a large degree of (read-write) false sharing and lots of off-node accesses.

3. Data Reordering

We describe two data reordering methods and how to apply them to the two categories of irregular applications described above. Each method consists of two phases: first, it constructs a sorting key for every object (a particle, a mesh point, etc.) and sorts the keys to generate the rank; second, the actual objects are reordered according to the rank. Since the second step is the same for all reordering methods, we focus on different ways of generating sorting keys. Figure 3 shows the two types of ordering: space-filling curves (Morton or Hilbert) and row or column ordering.

3.1 Space-Filling Curves

Morton [27] or Hilbert [17] space-filling curves create a linear ordering of points in a higher-dimensional space that preserves the physical adjacency of the points. The Morton ordering is achieved by constructing keys for sorting the subdomains by interleaving the bits of the subdomain coordinates. For Hilbert curves, algorithms based on bit manipulation [5] and finite-state diagrams [3] exist. An elegant recursive algorithm can be found in [18]. We focus on Hilbert reordering because it traverses only contiguous subdomains and thus potentially results in better data locality in the reordered data structure.

3.2 Column and Row Ordering

The sorting keys for row or column ordering are generated by simply concatenating the bits of the z–, y–, and x–coordinates of data units. For column ordering, z-coordinates form the least significant bits, and for row ordering, x-coordinates form the least significant bits.

3.3 Category 1 Applications

To combat poor spatial locality resulting from the mismatch between the random data ordering of the main data structure and the physical locality desired by the computation partition, we reorder the main data structure once using some space-filling curve. Going back to the example from Section 2, the locations of the particles to be updated by each processor after Hilbert reordering are shown in Figure 4. Comparing Figure 4 with Figure 1, one can see that most particles in the same pages will be updated by the same processors. The effectiveness of this data reordering on reducing false sharing can be dramatic. For example, if we reorder the particle array in the Barnes-Hut benchmark described in Section 2.1 using the Hilbert ordering, the degree of false sharing on the same 384 pages containing the 32768 particles is drastically reduced, as shown in Figure 5. On 16 processors, the average number of processors sharing a page is reduced from 9.5 to 3.

3.4 Category 2 Applications

For irregular applications in Category 2, reordering the main data structure using space-filling curves does not always produce the best result. Recall from the discussion in Section 2 that because of the block partitioning of the object array, these applications already exhibit good spatial locality for writes. The goal of reordering here is to improve the spatial locality for reads. To do so, we need to co-locate physically close particles as before, because that makes most of the interactions through the interaction list local. Having done that, however, we also want, for each processor p, to reduce the number of consistency units containing particles assigned to other processors but on the interaction lists of particles assigned to p.

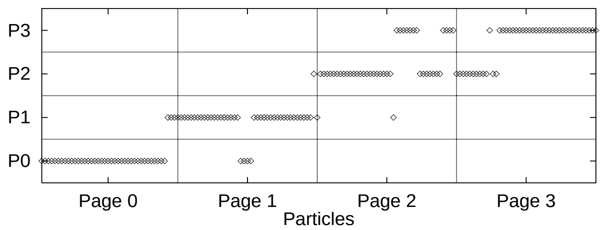

Figure 4: Locations of the 168 particles in each of the four 4-kilobyte pages to be updated by the four processors after Hilbert reordering.

Intuitively, row or column reordering tends to partition the 3-D space into slices, whereas space-filling curves tends to partition the space into cubes (see Figure 6). As a result, with column reordering, the objects on the partner list of the objects assigned to processor p tend to be assigned to fewer processors than with Hilbert reordering. For large consistency units, as in page-based software shared memory systems, column ordering produces the best results, because the objects on the interaction list tend to be on a few consistency units on these neighboring processors. In contrast, with Hilbert reordering, consistency units need to be fetched from a larger number of processors. For smaller consistency units, as on hardware shared memory systems, the larger surface of the slice produced by column reordering compared to the surface of the cube generated by Hilbert reordering produces the opposite effect. Here, column reordering spreads out the objects on the interaction list over more consistency units, and therefore Hilbert reordering works better.

3.5. Programming Interface

Column (or row) ordering is trivial to generate. The space-filling curves are more involved to compute, even though the Hilbert [17] space-filling curves have been known for over a century, and algorithms based on bit manipulation [5] have existed for almost three decades. We therefore implement both reorderings as library functions. Both functions consists of three steps: generating keys,

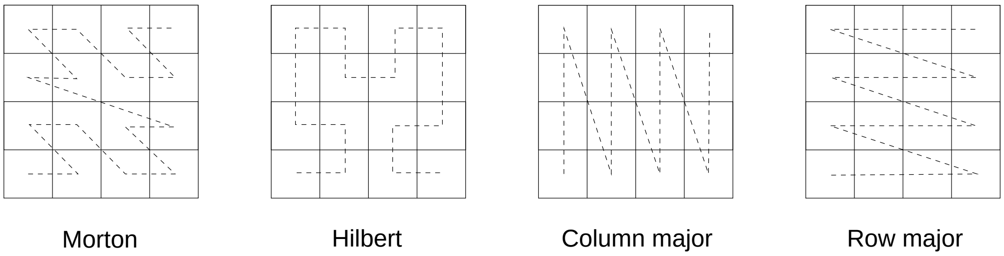

Figure 3: Data reordering methods: space-filling curves (Morton and Hilbert) and column or row major ordering.

ranking the keys, and reordering data according to the ranks. Including all three steps, the Hilbert curve reordering function is about 100 lines of C code. The data reordering functions provided are sequential, and can be called by a single processor as often as necessary.

In order to implement data reordering primitives that are applicable to arbitrary data types, the interface to each function includes a pointer to the object array, the size of an object, the number of objects, the dimensionality of the coordinates, and a function for obtaining the ith coordinate of the nth element. Specifically, the C interfaces to the two reordering primitives are:

```
void column_reorder(void *object,
                    int object_size,
                    int num_of_objects,
                    int num_of_dim,
                    double (*coord)(...));
void hilbert_reorder(void *object,
                    int object_size,
                    int num_of_objects,
                    int num_of_dim,
                    double (*coord)(...));
```

The following code segment shows the original data structure for particles in Barnes-Hut and the added code for calling the hilbert reordering function.

```
/* struct definition for a body */
typedef struct {
    short type;
    real mass;
    double pos[3];
    ...
} body, *bodyptr;

bodyptr bodytab;

/* program provided coord function */
double coord(bodyptr btab, int i, int dim)
{
    return btab[i].pos[dim];
}

main()
{
  /* serial initialization of particles */
  hilbert_reorder(bodytab, sizeof(*bodytab),
                  num_bodies, 3, coord);
  /* parallel executions */
}
```

4. Experimental Environment

4.1. Platforms

We evaluate the effectiveness of the data reordering techniques presented in this paper on both software distributed shared memory systems and the newest generation of hardware shared memory machine – the SGI Origin 2000.

4.1.1 SGI Origin 2000

Our hardware shared memory platform is an SGI Origin 2000 [23] with a total of 16 300Mhz MIPS R12000 processors, configured as two boxes of 4 nodes each. The two boxes are connected with a CrayLink. Each node consists of two processors. Within a node, each processor has separate 32KB first level instruction and data caches, and a unified 8MB of second-level cache and 128 byte block size. The machine has 10GB of main memory and a 16KB page size. In our experiment, the detailed measurements of the number of L2 cache misses and the number of TLB misses are collected using the processor event counter provided by the MIPS R12000 processors.

4.1.2 Software DSMs

We use the TreadMarks system [1] from Rice and a modified version of TreadMarks that implements Princeton's home-based LRC (HLRC) protocol [35]. Both systems are page-based and use multiple-writer protocols and lazy release consistency to alleviate the worse effects of false sharing. The two protocols differ in the location where modifications are kept and in the method by which they get propagated. A detailed comparison between the two protocols can be found in [10].

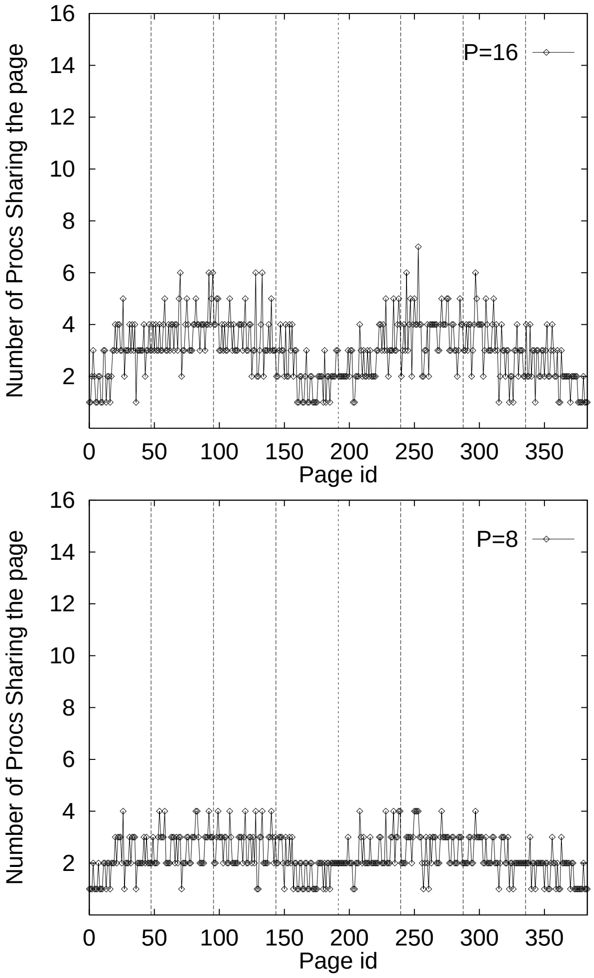

Figure 5: Number of processors sharing each of the 384 pages for the particle array of Barnes-Hut with reordering particles with 32K particles on 8 and 16 processors.

Both DSM systems run on a switched, full-duplex 100Mbps Ethernet network of 16 300 MHz Pentium II-based computers. Each computer has a 512K byte secondary cache and 256M bytes of memory. All of the computers were running FreeBSD 2.2.6 and communicating through UDP sockets. On this platform, the round-trip latency for a 1-byte message is 126 microseconds. The time to acquire a lock varies from 178 to 272 microseconds. The time for an 16-processor barrier is 643 microseconds. The time to obtain a diff varies from 313 to 1,544 microseconds, depending on the size of the diff. The time to obtain a full page is 1,308 microseconds. All of the above basic performance numbers are measured using a set of microbenchmarks which are part of the TreadMarks application suite, one for measuring each operation. In our experiment, we measure the total number of messages and total amount of data transmitted by the different versions of each application running on the two software DSM systems.

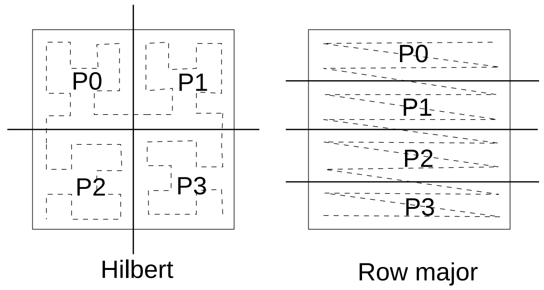

Figure 6: Boundary points using the Hilbert ordering are likely to be in more pages than using row major ordering in Moldyn.

Application	Size/Iter.	Sync.	Object Size (bytes)
Barnes-Hut	65536, 6	b	104
Fast Multipole Method	65536, 3	b,l	104
Water-Spatial	32768, 10	b,l	680
Moldyn	32000, 40	b	72
Unstructured	mesh.10k, 40	b,l	32

Table 1: Applications, input data sets, synchronization (l=lock, b=barrier), and data object sizes.

4.2. Applications

We used five irregular applications in this study: Barnes-Hut with sequential tree building, Fast Multipole Method, and Water-Spatial from the SPLASH-2 benchmark suite [33], Moldyn and Unstructured from the Chaos benchmark suite [11]. [2]

Table 1 summarizes the relevant characteristics of the applications. For each application, it lists the problem size, the method of synchronization (locks, barriers, or both) and the data object sizes. Intuitively, the overhead of reordering can become higher as we increase the problem size, as the cost of reordering increases as $n\log(n)$ and thus grows either faster than or at the same rate as the computation costs of all the above applications grow. To show a somewhat 'worse' case scenario, we have chosen the above problem sizes to be relatively large compared to typical problem sizes used in previous performance studies of shared memory systems.

5. Results

For each combination of platform and application, we present the results for the *original* version and for the *re-*

[2] In their original form, the Chaos benchmarks are message-passing programs that use the Chaos collective communication library. We ported them to our shared-memory platforms.

ordered version. For category 2 applications, we provide results both for Hilbert and column ordering.

All speedups are computed relative to the single-processor version of the *original* benchmark. In computing the speedup of the *reordered* versions, we include the execution of the reordering routine in the overall execution time.

5.1. Overall results on the SGI Origin 2000

Figure 7 presents the speedups of the various versions on the SGI Origin 2000. Table 2 presents the total execution time (excluding data reordering time), the time spent in the data reordering routine, the number of L2 cache misses, and the number of TLB misses for both version of all benchmarks, on a single processor and on all 16 processors.

On 16 processors, compared with the original versions of the benchmarks, all benchmarks except Water-Spatial achieve benefits from reordering, ranging from 12.4% for Barnes-Hut to 99% for Unstructured.

Data reordering improves the performance even on a single processor of the Origin. Specifically, the execution times of the reordered versions are between 1.6% to 8.3% lower than those of the original versions for four out of the five benchmarks. For these four benchmarks, the improvement comes either from significant reductions in TLB misses, e.g., a factor of 9.15 for Barnes-Hut and a factor of 38.7 for FMM, or from a reduction in L2 cache misses, e.g., 4.6% for Moldyn (using Hilbert reordering) and a factor of 4.9 for Unstructured (using Hilbert reordering). Since on the CC-NUMA SGI Origin 2000, the L1 cache miss penalty is insignificant compared to the L2 cache miss penalty, the number of L1 cache misses is not reported. For Water-Spatial, there is no improvement in execution time since the object (molecule) size, 680 bytes, is much larger than the L2 cache line size. In fact, there is a slight increase in the L2 cache misses and TLB misses. This is because on a single processor, the traversal on the 3-D grids degenerates to column ordering, which conforms well with the initial molecular ordering from initialization.

Table 2 shows that for the four benchmarks which show improvement as a result of reordering, the additional improvements for 16 processors on top of those already seen on a single processor, come mainly from reduced L2 cache misses. Specifically, the reduction in L2 cache misses ranges between a factor of 2.0 for Barnes-Hut and a factor of 2.2 for FMM. Data reordering makes little difference for Water-Spatial. Since the data object size in Water-Spatial is much larger than the L2 cache line size, there is little false sharing regardless of how the data is ordered.

5.2. Overall results on software DSMs

Figures 8 and 9 show the speedup comparison among the various versions of each benchmark under TreadMarks and HLRC on 16 processors. Table 3 presents sequential execution time, parallel execution time (excluding reordering), execution time of the reordering routine, number of messages, and amount of data on 16 processors for the various versions of the benchmarks on TreadMarks and HLRC.

On TreadMarks, the reordered versions of the benchmarks achieved from 30% to 366% better speedups than the original versions. On HLRC, the reordered versions of the benchmarks achieved from 14% to 269% better speedups than the original versions. In both cases, Moldyn benefited the least and FMM the most.

The improvements in the reordered versions of the benchmarks come from reduced false sharing, which results in dramatic reductions in both the amount of data transmitted and the number of messages. On TreadMarks, compared with the original versions, the reordered versions of the benchmarks send from 2.0 (Moldyn) to 3.7 (FMM) times less data, and from 1.4 (Unstructured) to 12.3 (Barnes-Hut) times fewer messages.

On HLRC, compared to the original versions, the reordered versions of the benchmarks send from 1.2 (Moldyn) to 5.0 (FMM) times less data, and from 1.4 (Moldyn and Unstructured) to 3.5 (FMM) times fewer messages.

The reason reduced false sharing results in a larger performance improvement in TreadMarks than in HLRC is because compared to in HLRC, TreadMarks sends many more messages (though with the same amount of total data) for the same degree of false sharing. The detailed explanation for this can be found in [10].

5.3. Detailed Discussions

5.3.1 Category 1 Applications

Barnes-Hut simulates the evolution of a system of particles under the influence of gravitational forces, as described in detail in Section 2.1. It is modified from the SPLASH-2 Barnes-Hut benchmark by changing the parallel tree building step into a sequential tree building phase. The code implements a hierarchical (also called tree-based) N-body algorithm. Reads and writes are to individual particles and thus occur with very fine granularity. The high degree of false sharing comes from the mismatch between the random ordering of particles in the shared particle array and the spatial locality among particles accessed by each processor. The mismatch is largely removed by a Hilbert reordering of the particles in the particle array which to a large extent brings particles updated by the same processor to the same pages.

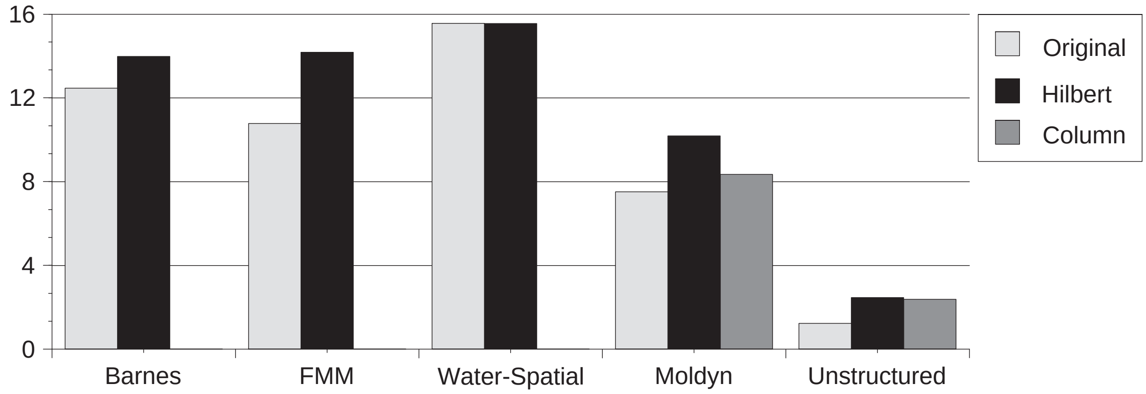

Figure 7: Speedup comparison among random, original, and reordered versions of the applications on Origin 2000 with 16 processors.

Application	Version	Cost of Reorder (sec.)	1 Processor			16 Processors		
			Time (sec.)	L2 Cache Misses	TLB Misses	Time (sec.)	L2 Cache Misses	TLB Misses
Barnes-Hut	original		129.3	457267	50041379	10.4	2284706	50273087
	reordered	0.26	117.2	420442	5469307	8.99	1148159	5705782
Fast Multipole Method	original		69.3	9177722	21940902	6.44	6615243	22626223
	reordered	0.28	63.7	9014517	567241	4.61	3049778	699619
Water-Spatial	original		235.8	9754036	198319	15.2	2170170	754381
	reordered	0.47	235.9	11493221	370242	15.1	1893316	249131
Moldyn	original		33.7	12070308	113206	4.49	6061241	134070
	hilbert	0.09	31.9	11542334	116095	3.31	2969676	134301
	column	0.03	33.4	11213694	74436	4.04	3923060	108009
Unstructured	original		34.7	18186	35745	28.2	55770320	302240
	hilbert	0.06	34.1	3697	33432	14.1	26290356	253465
	column	0.05	34.7	11636	36037	14.6	28190625	245782

Table 2: Total execution time (excluding reordering time), the execution time of the reordering routine, the number of L2 cache misses and the number of TLB misses for various versions of the benchmarks on a single processor and on 16 processors of the Origin 2000.

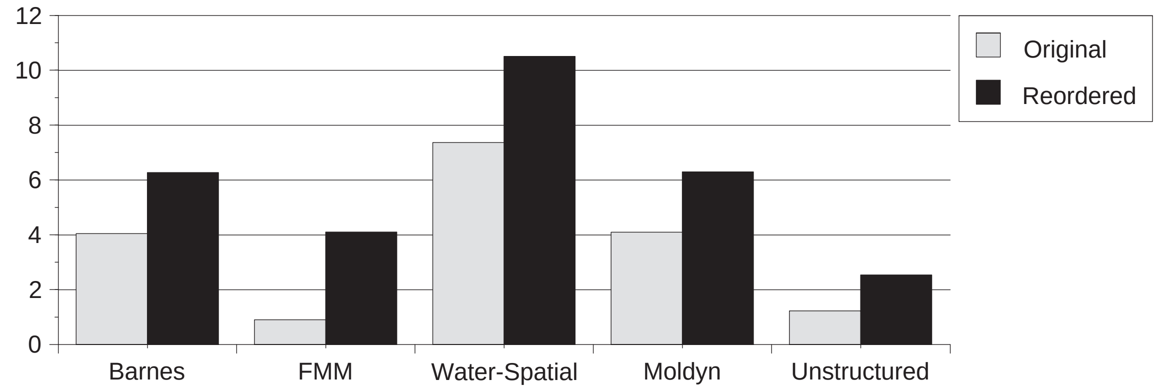

Figure 8: Speedup comparison among the various versions of the benchmarks on TreadMarks on 16 processors.

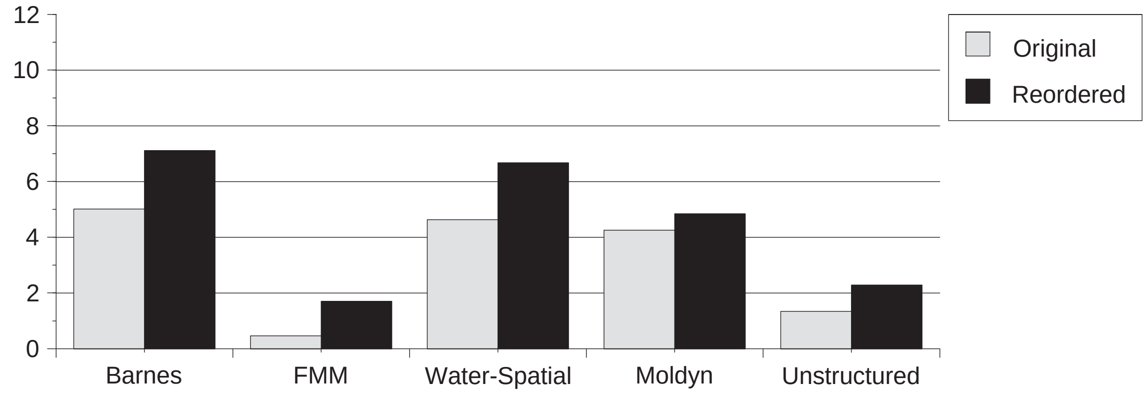

Figure 9: Speedup comparison among the various versions of the benchmarks on HLRC on 16 processors.

Fast Multiple Method from SPLASH-2, like Barnes-Hut, also simulates the evolution of a system of particles under the influence of gravitational forces. However, it simulates interactions in two-dimensions and uses a different hierarchical n-body algorithm called the adaptive Fast Multipole Method [16]. It has the same major data structures as Barnes-Hut. Namely, it has particles and tree cells, but differs from Barnes-Hut in that the tree is not traversed once per particle. There is only a single upward traversal followed by a single downward traversal which together compute interactions between the cells and propagate their effects down to each leaf cell. The effect on each leaf cell is then applied to the particles inside that cell.

The false sharing in FMM is similar in nature to that in Barnes-Hut. On one hand, computation (i.e. the tree) is partitioned using some space-filling curve and has good spatial locality in the tree cells, which are created independently by the processors and stored in some per-processor (though shared) arrays. On the other hand, the particles are generated and stored in random order in the shared array without any locality between particles adjacent in space. This creates false sharing at the various stages of FMM that access particles. Our solution is to reorder particles in the particle array using the Hilbert ordering. Table 4 shows that the improvement in the reordered version compared with the random or original versions is seen at various steps that access the particles.

Version	Original	Reordered
Build tree	6.78	0.76
Build List	2.51	2.53
Partition	0.40	0.57
Tree traversal	13.3	1.60
Inter particle	25.1	6.70
Intra particle	1.59	0.07
Other	8.16	0.38
Total	61.8	13.3

Table 4: Breakdown of time (in seconds) for the three versions of FMM in TreadMarks.

Water-Spatial from SPLASH-2 is a short-range N-body simulation code that evaluates forces and potentials in a

Application	Version	Seq. Time (seq.)	Cost of Reorder (sec).	TreadMarks Time (sec.)	TreadMarks Data (Mbytes)	TreadMarks Messages	HLRC Time (sec.)	HLRC Data (Mbytes)	HLRC Messages
Barnes-Hut	original	334.6		58.0	605.0	3132940	46.8	864.8	401864
	reordered		0.57	37.4	268.6	254304	33.0	324.6	142682
Fast Multipole Method (16K for HLRC, 11.5 sec.)	original	54.1		60.2	548.5	2693104	25.0	808.3	586870
	reordered		0.68	13.2	149.2	360485	6.77	161.0	167519
Water-Spatial	original	780.0		106.0	1310.	1261510	168.5	2397.	1018155
	reordered		0.97	70.5	636.5	428450	117.0	1125.	411725
Moldyn	original	99.1		24.2	708.9	435872	23.3	682.0	343755
	column		0.13	15.8	347.6	244526	20.5	588.9	241431
	hilbert		0.24	44.4	939.6	1350624	57.8	1637.	817931
Unstructured	original	182.0		148.6	3608.	2749052	136.0	3582.	2637236
	column		0.12	71.8	1487.	1955688	79.6	2008.	1924864
	hilbert		0.16	84.6	1426.	2019822	102.6	2288.	2063634

Table 3: Sequential execution time, and parallel execution time (excluding reordering), execution time of the reordering routine, number of messages, and amount of data on 16 processors for the various versions of the benchmarks on TreadMarks and HLRC.

system of water molecules over time. It imposes a uniform 3-D grid of cells on the problem domain. The use of grids to chain spatially adjacent molecules together allows a processor to only look at neighboring cells in order to find all molecules that are within the cutoff radius from the molecules in each cell owned by that processor. Molecules can move between neighboring cells at each iteration. The false sharing is caused by the mismatch between the random ordering of molecules in the shared address space and the locality-aware 3-D partition of the 3-D grids. Reordering the molecules using the Hilbert ordering greatly reduces the false sharing.

5.3.2 Category 2 Applications

Moldyn is a molecular dynamics simulation ported from the Chaos benchmark suite. Its computational structure resembles the non-bonded force calculation in CHARMM [4]. Non-bonded forces are long-range interactions existing between each pair of molecules. CHARMM approximates the non-bonded calculation by ignoring all pairs which are beyond a certain cutoff radius. The cutoff approximation is achieved by maintaining an *interaction list* of all the pairs within the cutoff distance, and iterating over this list during each time step. The interaction list is used as an indirection array to identify interacting partners. Since molecules change their spatial location every iteration, the interaction list is periodically updated. Moldyn belongs to Category 2 since molecules are stored in a 1-D array and computation is partitioned by simply dividing the molecule array evenly among the processors. The short-range interaction requires each processor to read and write any molecules adjacent to the molecules that it owns. Since molecule reordering is not restricted by any computation partitioning as in Category 1 applications, the task of reducing false sharing boils down to reducing the number of pages containing such “neighboring” molecules. The solution is to reorder the molecules. On software DSMs, due to the large (page) coherence units, reordering molecules using column ordering is almost a factor of 3 times better in TreadMarks and HLRC than using Hilbert ordering. On an Origin 2000, due to the much smaller (cache line) coherence units, Hilbert reordering results in 22% better speedup than column reordering.

Unstructured from the Chaos benchmark suite is a simplified version of a computational fluid dynamics (CFD) application. It employs the finite element method, which decomposes a physical structure into an unstructured mesh. The mesh is represented by nodes, edges that connect two nodes, and faces that connect three or four nodes. The mesh is static, so its edge and face connectivities do not change. Since an unstructured mesh is essentially a decomposition of the physical domain, the edges and faces follow the physical adjacency of nodes. In other words, edges or faces only connect physically adjacent nodes. The computation contains a series of loops that update nodes by iterating over nodes, or perform interactions between connected nodes by iterating over the edges. In our shared memory version of this benchmark, the iterations on the nodes or edges are partitioned among processors. Thus the application falls into Category 2. To reduce the number of edges whose nodes belong to different processors, we reorder the nodes using column ordering or Hilbert ordering. Consistent with Moldyn, on software DSMs, column reordering outperforms Hilbert reordering and improves the speedup of the benchmark by a factor of 2.1 on TreadMarks and a factor of 1.7 on HLRC, compared to the original versions. On the Origin 2000, Hilbert reordering outperforms column reordering and improves the speedup of the original version by a factor of 2.0.

6. Related Work

In message passing programs, the data is *explicitly* partitioned and then allocated in the (disjoint) address spaces of the corresponding processes. Whereas, our data reordering techniques achieve an *implicit* partitioning of the data. Both are a means to the same end: a reduction in the amount of communication.

There have been both compile-time and run-time schemes to reduce false sharing in shared memory programming. Compile-time techniques use loop transformation [15] or data structure padding [19]. These techniques are limited by the analysis that the compiler can perform. No such limitations exist for the library-based data reordering that we use. Run-time techniques mainly use small consistency units or consistency units adapted to the data structures used in the computation [28, 34]. These systems lose the aggregation effect of large consistency units, which our system maintains while still considerably reducing false sharing.

Jiang et al. [22, 21, 20] have studied how to re-structure some of the SPLASH-2 benchmarks for software DSM on networks of uniprocessors and multiprocessors, and on large hardware DSMs. Their restructuring techniques fall into three categories: padding and alignment, data structure changes, and algorithmic changes. Padding and alignment cause only limited fragmentation on hardware shared memory machines, but for page-based software shared memory, where a page can be 100 times larger than a data object, the fragmentation cost quickly becomes prohibitive. Their data structure changes consist mainly of changing 2-D arrays into 4-D arrays to block data elements in dense matrix computations. While similar in its goals to our data reordering, the applications that benefit from it are different. The algorithmic changes are by nature application-specific and require extensive knowledge about the applications. Our data reordering techniques are less intrusive in terms of source code modifications and require less knowledge of the application. We intend to study whether some of their proposed optimizations can be combined with data reordering for additional performance improvement.

Data locality has long been recognized as one of the most significant performance issues for modern uniprocessor architectures. There has been a large body of research on loop transformations for dense matrix codes, e.g. [14, 32, 7, 31]. Recently, several works have focused on array and pointer-based data structures in irregular applications. Ding and Kennedy [12] and Mellor-Crummey et al. [24] looked at irregular applications which perform irregular accesses of array elements via interaction (indirection) arrays. Such applications fall into our Category 2. The basic idea in Ding and Kennedy [12] is to examine the contents of indirection array and generate a new ordering for the array elements based on the access affinity implied by the indirection array. They then reorder data in memory and adjust the contents of the indirection arrays. The technique does not require the geometric coordinates of the physical quantities that array elements are modeling. Like our work, Mellor-Crummey et al. [24] use of space-filling curves to reorder data and/or computation, but they focus on the uniprocessor memory hierarchy.

There have been several semi-automatic data placement approaches to improving cache performance on uniprocessors for heap objects. Calder *et al*. proposed cache conscious data placement [6]. They use profile information to find temporal locality between data objects, and reorder them in memory accordingly. Chilimbi et al. improve the performance of trees by clustering parent and child nodes in memory [9]. They developed a tree optimization routine to adjust the memory layout of a tree for improved locality for a given traversal, and a memory allocator that allocates memory close to a user specified heap location. They also proposed two techniques, *structure splitting* and *field reordering*, that improve temporal locality by changing the internal organization of fields of a data structure [8]. These optimizations are targeted at applications with data structures larger than a cache block.

7. Conclusions

This paper has demonstrated that data reordering can achieve substantial improvements in the performance of fine-grained irregular applications on shared memory systems. These data reordering techniques attempt to co-locate in memory the data that represents physically close objects. The result is improved spatial locality and reduced false sharing. Hilbert reordering is appropriate for hardware shared memory machines and for applications with sophisticated computation partitioning on page-based software shared memory systems. Column reordering works better for block-partitioned computations on software shared memory systems. We have provided a library which implements both reordering methods, and we have demonstrated that it can be used with very little modification to the benchmark programs. These libraries are also independent of the specific platform used. The revised programs should serve as better benchmarks for this class of applications on shared-memory machines.

Acknowledgement

We thank Doug Moore for contributing a highly optimized Hilbert ordering code.

References

[1] C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, Feb. 1996.

[2] J. Barnes and P. Hut. A hierarchical $O(N \log N)$ force calculation algorithm. *Nature*, 324:446–449, 1986.

[3] T. Bially. Space–filling curves: Their generation and their application to bandwidth reduction. *IEEE Trans. Inform. Theory*, IT-15:658 – 664, November 1969.

[4] B. Brooks, R. Bruccoleri, B. Olafson, D. States, S. Swaminathan, and M. Karplus. Charmm: A program for macromolecular energy, minimization, and dynamics calculations. *Journal of Computational Chemistry*, 4:187, 1983.

[5] A. R. Butz. Convergence with Hilbert's space filling curve. *J. of Computer and System Sciences*, 3:128 – 146, May 1969.

[6] B. Calder, C. Krintz, S. John, and T. Austin. Cache conscious data placement. In *Proceedings of the 8th Symposium on Architectural Support for Programming Languages and Operating Systems*, Oct. 1998.

[7] S. Carr, K. McKinley, and C.-W. Tseng. Compiler optimizations for improving data locality. In *Proceedings of the 6th Symposium on Architectural Support for Programming Languages and Operating Systems*, Oct. 1994.

[8] T. Chilimbi, B. Davidson, and J. Larus. Cache-conscious structure definition. In *Proceedings of the ACM SIGPLAN 99 Conference on Programming Language Design and Implementation*, May 1999.

[9] T. Chilimbi, M. Hill, and J. Larus. Cache-conscious structure layout. In *Proceedings of the ACM SIGPLAN 99 Conference on Programming Language Design and Implementation*, May 1999.

[10] A. L. Cox, E. de Lara, C. Hu, and W. Zwaenepoel. A performance comparison of homeless and home-based lazy release consistency protocols in software shared memory. In *Proceedings of the Fifth International Symposium on High-Performance Computer Architecture*, pages 279–283, Jan. 1999.

[11] 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, Sept. 1994.

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

[13] S. Dwarkadas, K. Gharachorloo, L. Kontothanassis, D. J. Scales, M. L. Scott, and R. Stets. Comparative evaluation of fine- and coarse-grain approaches for software distributed shared memory. In *Proceedings of the Fifth International Symposium on High-Performance Computer Architecture*, pages 260–269, Jan. 1999.

[14] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformation. *Journal of Parallel and Distributed Computing*, 5:587–616, Jan. 1988.

[15] E. Granston and H. Wijshoff. Managing pages in shared virtual memory systems: Getting the compiler into the game. In *Proceedings of the 1993 ACM International Conference on Supercomputing*, July 1993.

[16] L. Greengard. *The Rapid Evaluation of Potential Fileds in Particle Systems*. MIT Press, 1988.

[17] D. Hilbert. Über die stetige Abbildung einer Linie auf Flächenstück. *Math. Ann.*, pages 459–460, 1891.

[18] Y. C. Hu and S. L. Johnsson. Data parallel performance optimizations using array aliasing. In M. Heath, A. Ranade, and R. Schreiber, editors, *Algorithms for Parallel Processing*, pages 213–245. IMA Volumes in Mathematics and its Applications, Volume 105, Springer–Verlag, 1999.

[19] T. Jeremiassen and S. Eggers. Reducing false sharing on shared memory multiprocessors through compile time data transformations. In *Proceedings of the 5th Symposium on the Principles and Practice of Parallel Programming*, July 1995.

[20] D. Jiang, H. Shan, and J. Singh. Application restructuring and performance portability on shared virtual memory and hardware-coherent multiprocessors. In *Proceedings of the 6th Symposium on the Principles and Practice of Parallel Programming*, June 1997.

[21] D. Jiang and J. Singh. Scalability of home-based shared virtual memory on clusters of SMPs. In *Proceedings of the 1999 International Conference on Supercomputing*, May 1999.

[22] D. Jiang and J. Singh. Scaling application performance on a cache-coherent multiprocessor. In *Proceedings of the 26th International Symposium on Computer Architectures*, May 1999.

[23] J. Laudon and D. Lenoski. The SGI origin: a CCNUMA highly scalable server. In *Proceedings of the 24th International Symposium on Computer Architectures*, June 1997.

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

[25] 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 5th Symposium on the Principles and Practice of Parallel Programming*, July 1995.

[26] D. O'Hallaron, J. Shewchuk, and T. Gross. Architectural implications of a family of irregular applications. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, Jan. 1998.

[27] H. Samet. *Design and Analysis of Spatial Data Structures*. Addison–Wesley, 1990.

[28] D. Scales, K. Gharachorloo, and C. Thekkath. Shasta: A low overhead software-only approach for supporting fine-grain shared memory. In *Proceedings of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems*, Oct. 1996.

[29] J. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared-memory. Technical Report CSL-TR-91-469, Stanford University, Apr. 1991.

[30] J. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared-memory. *Computer Architecture News*, 20(1):2–12, Mar. 1992.

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

[32] M. E. Wolfe and M. S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN 91 Conference on Programming Language Design and Implementation*, June 1991.

[33] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, June 1995.

[34] M. Zekauskas, W. Sawdon, and B. Bershad. Software write detection for distributed shared memory. In *Proceedings of the First USENIX Symposium on Operating System Design and Implementation*, pages 87–100, Nov. 1994.

[35] Y. Zhou, L. Iftode, and K. Li. Performance evaluation of two home-based lazy release consistency protocols for shared virtual memory systems. In *Proceedings of the Second USENIX Symposium on Operating System Design and Implementation*, pages 75–88, nov 1996.