# I/O Performance Improvement through the Use of Compiler-Driven Memory Management

**Jenny Feng**
Computer Science Department
University of Houston
Houston, TX 77204, USA
E-mail: jennyfeng@yahoo.com

**Wendy Zhang**
Department of Computer Information Systems
Southern University at New Orleans
New Orleans, LA 70126, USA
E-mail: wzhang@suno.edu

and

**Ernst L. Leiss**
Computer Science Department
University of Houston
Houston, TX 77204, USA
E-mail: coscel@cs.uh.edu

## Abstract

A compiler combined with a user level runtime system can replace and outperform standard virtual memory management (VMM) for out-of-core problems. The execution time for single code and the overall impact on system resource utilization is lower for the compiler/runtime system combination than for VMM. Also this system does not require modification of the operating system kernel. In this paper we present a modified Comanche (Compiler Managed Cache) system and give new methodologies for providing efficient out-of–core programming without VMM for wavefront data access patterns. The experimental data show that Comanche performs better than VMM for the test case patterns under two standard operating systems, Windows and Linux, and has significantly less impact on system resources.

**Keywords:** I/O Management, VMM, Out-of-core, Compiler, Buffer

# 1 Introduction

Today’s high performance computer systems provide users with immense computational power to facilitate research across a wide spectrum of scientific disciplines. Reducing program execution time is commonly the reason for purchasing new, faster processors. However, coupled with the ability to perform high-speed computation, there is the needs to store, organize, access, distributed and visualize these data, resulting in I/O demands on local systems and communication across networks [4].

When the data sets required by these applications exceed the capacity of main memory, the computation becomes an out-of-core computation. Processing out-of-core data requires staging data in smaller granules that fit in main memory. Data required for the entire computation have to be fetched from files on disk so that improving disk I/O becomes extremely important. The speed gap between processors and disks continues to increase as VLSI technology advances at a tremendous rate while disk technology lags behind. As a result, disk I/O has become a serious bottleneck for many high performance computer systems. Hence, it is critically important to be able to construct I/O minimal programs [11].

Much research has been done on virtual memory management (VMM) and other related operating systems (OS) concepts, I/O subsystem hardware, and parallel file systems. Each of those approaches contributes to some degree to I/O performance, but they all lack *a global view of application behavior*, which limits their effectiveness.

Parallel I/O is a cost-effective way to address some I/O issues. The wide availability of inexpensive powerful PC clusters with high-speed networks makes parallel I/O a viable approach. Parallel I/O subsystems have increased the I/O capabilities of parallel machines significantly but much improvement is still needed to balance the CPU performance. The variety (private disks, shared disks or a combination of both) in the I/O architectures makes it difficult to design optimization techniques that reduce the I/O cost. The problem has become more severe since the size and complexity of applications have increased tremendously [13].

OS designers offer the handling of I/O activity via virtual memory management (VMM). Research on this approach considers the use of smart virtual memory, techniques which reshape the data reference patterns to exploit the given hardware facilities and system software, and replacement policies. Overall these techniques assume a considerable amount of help from the hardware.

A number of run-time libraries for out-of-core computations and a few file interfaces have been proposed, among them SIO [12], MPI-IO [2], and an extension of the traditional Unix file I/O interface for handling the parallel accesses to parallel disk subsystems [10]. The parallel file systems and run-time libraries for out-of-core computations provide considerable I/O performance, but they require much effort from the user; also they are not portable across a wide variety of parallel machines with different disk subsystems.

The difficulty of handling out-of-core data and writing an efficient out-of-core program limits the performance of high performance computers. Execution of some out-of-core programs does not perform well when they rely on the virtual memory management (VMM) system. There is a clear need for compiler directed explicit I/O for out-of-core computations [1] [3] [5] [6] [7] [8] [14] [15] [16] [17] [18] [19].

In this paper, we concentrate on the compiler-based approach to the I/O problem. The main rationale behind this approach is the fact that the compiler has unique information about the data needs of the program. The compiler can examine the size and shape of the data and the overall access pattern of the application. Compiler driven I/O management should generate code to restructure out-of-core data, computation, and the management of memory resources. A compiler combined with a user level runtime system can replace and outperform standard virtual memory management for out-of-core problems [14]. A compiler with a good combination of file layouts on disks and loop transformations is successful at optimizing programs, which depend on disk-resident data in distributed-memory machines [9].

Comanche (an acronym for COmpiler MANaged caCHE) is a compiler run time I/O management system [14]. It is a compiler combined with a user level runtime system. More details about Comanche will be given in Section 2.

It was observed during initial testing of Comanche that memory mapping a large file has a significant impact on system resources. We are mainly interested on how to access large data sets instead of how to manipulate these data after we get them. From our experiments, a major part of the execution time is spent on *I/O* instead of on calculation. Once we get the data, relatively little time is needed for doing calculations based on the data. The out-of-core *WaveFront* problems used in this paper are used to present the benefit of our modified Comanche *API* which provides data access methods for cases more general than the original version. In Section 3, we discuss the *seeker/reaper* paradigm and show how to apply it to minimize the amount of space required to carry out the computation. In Section 4, we will show the results of our experiments on two workstations, one running Windows98 and another running Linux 5.5. In Section 5, we draw conclusions from the results of our study.

## 2 Comanche

Comanche (an acronym for COmpiler MANaged caCHE) is a compiler run time I/O management system [13]. It is a compiler combined with a user level runtime system and effectively replaces virtual memory management by allowing direct control over which pages are retained in the active memory set. The current Comanche system is running under the RedHat 5.5 Linux operating system on a PC with an Intel PentiumPro (133Mhz or faster) processor and 48MB or more RAM. The system is also running under Window98 with an Intel PentiumPro 500Mhz processor and a RAM of 96MB.

The standard entity in the Comanche runtime system is a two-dimensional array. Higher dimensions can be supported, or else the accesses can be translated to two-dimensional accesses. Data are assumed to be in row major layout on disk. The ideal array is a square matrix.

There are two structures declared in the Comanche system. One (subrow_s) is a block structure used to buffer a sub-row of data, the other structure is a matrix structure (array_s) that holds the information of the matrix on disk and has buffers to hold several sub-rows in memory.

```
typedef structure subrow_s  {
       double *data;
       int wflag;
       int refcnt;
       int subrowindex;
  } Row;


typedef struct array_s {
       int nrows, nelems, elesize;
       int nsubrows, nsubelems;
       FILE *fp;
       long offset;
       char *name;
       int *map;
       int nbuffs, buffsize;
       Row **buffers;
       int victim, wflag, total_mem;
       } Matrix;
```

Besides these structures, there are several functions in Comanche. Two major functions are *comanche_attach* and *comanche_release* which are used to perform the sub-row mapping.

When a data set is too large to fit into memory, VMM will map the array into a one-dimensional vector and then that vector is cut up into sub-arrows. Comanche will take a row of an array and cut it up into sub-rows. Through the use of the functions provided by Comanche, the I/O behavior is under the control of the resulting out-of-core program.

The *comanche_attach* and *comanche_release* functions tell the runtime system that the block is to be mapped into memory; then an address to the cached data is to be returned. The runtime system will not reclaim memory that has been attached as long as it remains attached. It does not matter how much time has passed since the buffer was last referenced. The out-of-core program manages the duration of mapped data and ensures that the number of attach operations will not over-fill the available memory before the data's subsequent release.

## 3 WaveFront Problems

*WaveFront* stands for a square with tilted lines inside (as illustrated in Figure 3.1) the array A in waves $W_1$ through $W_{2n-1}$ as follows (assuming N is 4):

$W_1$ A(1,1)
$W_2$ A(2,1), A(1,2)
$W_3$ A(3,1), A(2,2), A(1,3)
$W_4$ A(4,1), A(3,2), A(2,3), A(1,4)

$W_5$ A(4,2), A(3,3), A(2,4)
$W_6$ A(4,3), A(3,4)
$W_7$ A(4,4)

In general, we have:
$W_i$ A(i,1), A(i-1,2),…,a(2,i-1),A(1,i) for i = 1 …N
$W_{N+j}$ A(N,j+1), A(N-1,j+2),…,A(j+2,N-1),A(j+1,N) for j =1..N-1

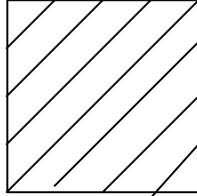

Figure 3.1 *WaveFront* Access Pattern

This test traverses along each tilted line, calculates the sum of all the elements on it, and writes the total value to a specified file. The original code is as follows:

```
void wavefront( int n, double *A, FILE *fp)
{
   int sum, column, row, position, i, count;
   double total;

   count = 2 * n - 1 ;

i=0;
for( sum=0; sum<n; sum++ )
{
  total = 0;
  for( column=sum; column>=0; column-- )
    {
      row = sum - column;
      position = row*n + column;
      total += A[position];
    }
  /* write result back to file */
  assert( fwrite(&total, sizeof(double), 1, fp) == 1 );
  i++;
}
for( sum=n; sum<2*n-1; sum++ )
  {
    total = 0;
    for( row=sum-(n-1); row<n; row++ )
    {
      column = sum - row;
      position = row*n + column;
      total += A[position];
    }
    /* write result back to file */
    assert( fwrite(&total, sizeof(double), 1, fp) == 1 );
    i++;
  }
}
```

In order to calculate the sum of each tilted line, every element in it has to be accessed. The tilted line near the center of the square may grow beyond the size of a single page. The closer the titled line to the center, the more page fault may occur. Therefore, we want to keep all the elements on the current titled line in the buffers all the time.

We use a simple optimization based on a *seeker/reaper* paradigm. A *seeker* is always one step ahead of the real attach operation; its responsibility is to find which subrow will be referenced, and attach this subrow to the buffers. A *reaper* always follows the *seeker;* its responsibility is to find which subrow will not be referenced any more, and release this subrow before the *seeker* attaches other subrow. We divided the buffers into several grouped; each group may have many subrows among which the first subrow is the *seeke*r, and the last subrow is the *reaper.* Every group has a pair consisting of a *seeker* and a *reaper*. All the current used data are stored in the *buffers.* By swapping useless data out and swapping useful data in, we can guarantee that all the currently used data are in their *buffers* and the *buffers* are not overfilled.

Before the main loop is executed, the *seeker* must attach the initial reference subrows. After the main loop is completed, the *reaper* releases all the referenced subrows. During the execution of the main loop, the *reaper* also releases never-to-be-referenced subrows and leaves enough space in the *buffers* for *seeker* to attach a new subrow.

As illustrated in Figure 3.2, we initially have three groups of data attached in the *buffers*, buffer1, buffer2, and buffer3. Then we traverse each tilted line from left to right, calculate the sum of elements on each tilted line and write the result to a specified file. Row-faults do not occur until the tilted line is out of the range of buffer1 (line A). This scenario is illustrated in Figure 3.3. At this critical point, the *reaper* of buffer1 releases the earliest attached subrow, and its corresponding *seeker* attaches a new subrow. The buffer1 is moving one subrow forward (see Figure 3.4). Similarly, when the tilted line grows beyond buffer2 (line B) as illustrated in Figure 4.5, the *reaper* of buffer2 releases the current subrow, and the corresponding *seeker* attaches the new subrow. As long as the new subrow keeps coming up, the *reaper* of each *buffer* keeps releasing its earliest attached subrow and the corresponding *seeker* keeps attaching that new subrow. Therefore, the *buffers* are guaranteed to store the most recent data.

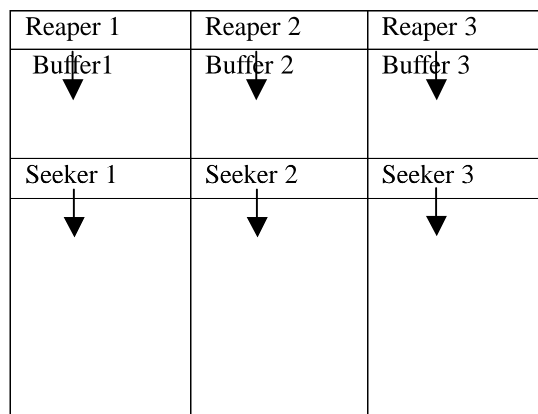


Figure 3.2 WaveFront – Step 1

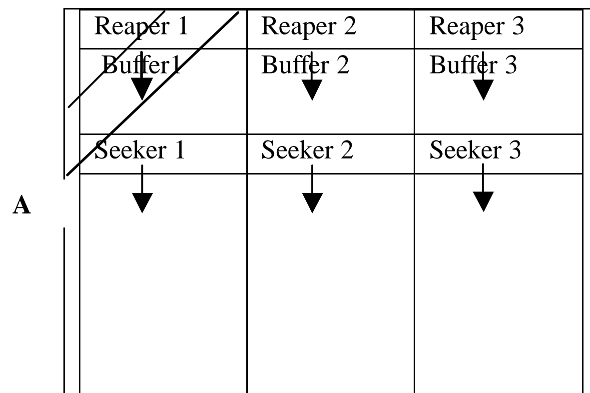


Figure 3.2 WaveFront – Step 2

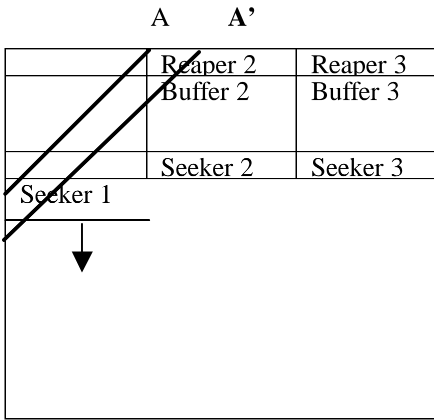


Figure 3.4 WaveFront – Step 3

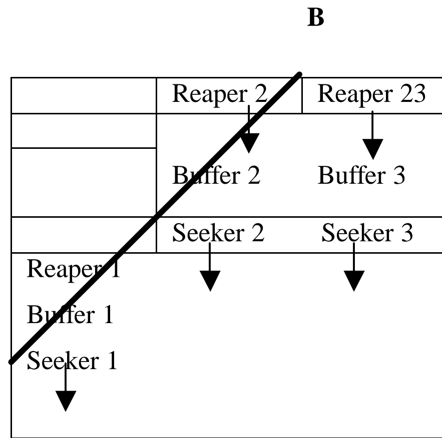


Figure 3.5 WaveFront – Step 4

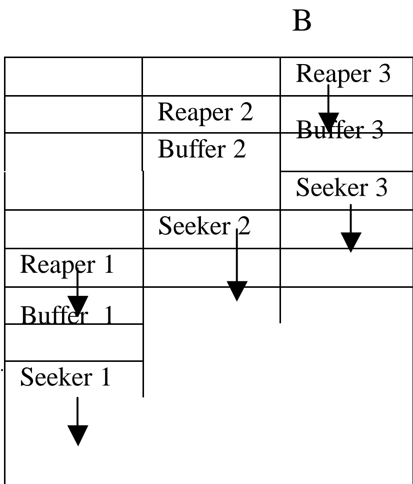


Figure 3.6 WaveFront – Step 5

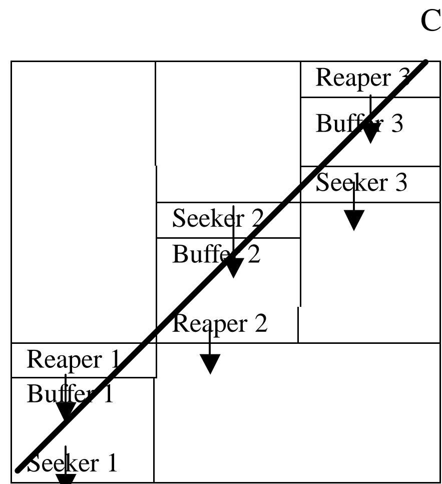


Figure 3.7 WaceFront – Step 6

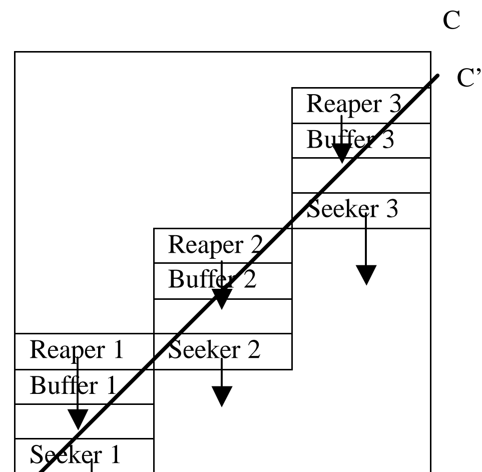


Figure 3.8 WaveFront – Step 7

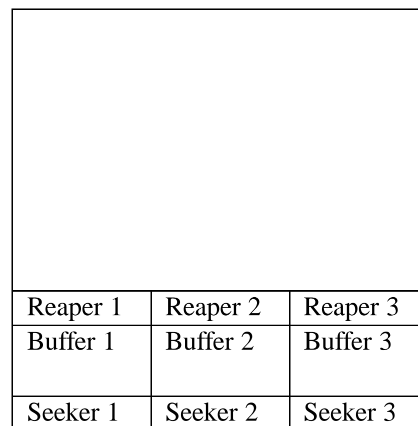


Figure 3.9 WaveFront – Final state

## 4 Experiments

We have implemented the algorithm suggested in Section 3 in an out-of-core program written in the C language. The C compiler generates working code using the Comanche runtime system. The code was run under the Window98 operating system with an Intel PentiumPro 500 MHz processor and 96MB RAM, and under the Redhat Linux5.5 operating system on a single processor PC with an Intel PentiumPro 133MHz microprocessor and 48MB RAM.

We have run one set of experiments for a double precision matrix of size 3600 x 3600 involved in the *WaveFront* application. The total data set space is 3600 x 3600 x 8 = 103.68MB. Another set of experiment was run for the size 4000 x 4000 with a total data set space of 4000 x 4000 x 8 = 128MB. Data files were initialized with random values in the in the interval (-1, +1). The code is given in Figure 4.1.

```
void wavefront( Array * aptr, FILE * outFp ) {
    int nrows, nelems, nsubrows, nsubelems, nbuffs;
    int * seeker, * reaper, chkn, n, start, end;
    int sum, column, row, position, subrow, subcolumn, i, k;
    double total = 0;
    double * A;
    nrows = aptr->nrows;
    nelems = aptr->nelems;
    nsubrows = aptr->nsubrows;
    nsubelems = aptr->nsubelems;
    nbuffs = aptr->nbuffs;
    n = nelems / nsubelems;
    /* assumption:
    each chunk is a square(nsubelems*nsubelems).  n = chkn */
    chkn = nbuffs / nsubelems;
    seeker = (int *)malloc(sizeof(int)*chkn);
    assert(seeker);
    reaper = (int *)malloc(sizeof(int)*chkn);
    assert(reaper);
    for( i=0; i<chkn; i++ )
    {
        start = i;
        end = start + n*nsubelems;
        reaper[i] = start;
        for( seeker[i]=start; seeker[i]<end; seeker[i]+=n )
        {
                comanche_attach(aptr, seeker[i], FALSE);
        }
        // seeker is always one step ahead of real comanche_attach
    }
    /* top-left triangle */
    for( sum=0 ; sum<nelems; sum++ )
    {
        total =0;
        for( column=sum; column>=0; column-- ){
                row = sum - column;
                position = row * nelems + column;
                subrow = position/nsubelems;
                subcolumn = position%nsubelems;
                i = subrow % n; // chunk #
```

```
            assert( i < chkn );
            if( subrow == seeker[i] ){
                  comanche_release(aptr, reaper[i]);
                  reaper[i] += n;
                  A = comanche_attach(aptr, seeker[i], FALSE);
                  total += A[subcolumn];
                  seeker[i] +=n;
            }
            else{
                  A= comanche_attach(aptr, subrow, FALSE);
                  total += A[subcolumn];
                  comanche_release(aptr, subrow);
            }
      }
      assert( fwrite(&total, sizeof(double), 1, outFp) == 1 );
   }
   /* bottom-right triangle */
   for( sum=nelems; sum<2*nelems-1; sum++ ){
      total = 0;
      for( row=sum-(nelems-1); row<nelems; row++ )
      {
            column = sum - row;
            position = row * nelems + column;
            subrow = position/nsubelems;
            subcolumn = position%nsubelems;
            i = subrow % n; // chunk #
            assert( i < chkn );
            if( subrow == seeker[i] ){
                  comanche_release(aptr, reaper[i]);
                  reaper[i] += n;
                  A = comanche_attach(aptr, seeker[i], FALSE);
                  total += A[subcolumn];
                  seeker[i] +=n;
            }
            else {
                  A = comanche_attach(aptr, subrow, FALSE);
                  total +=A[subcolumn];
                  comanche_release(aptr, subrow);
            }
      }
      assert( fwrite(&total, sizeof(double), 1, outFp) == 1 );
   }
   /* release all buffers */
   for( i=0; i<chkn; i++ )
   {
      for( k= reaper[i]; k<nsubrows; k=k+n )
         comanche_release(aptr, k);
  }
}
```

Figure 4.1 Comanche Code for *WaveFront*

**4.1 Single WaveFront application**

The first test was for the single out-of-core *WaveFront* application. For each test case, the result in Table 4.1 is the average of five tests. The time used for executing the whole program is measured in seconds. The ratio represents the

time of the virtual memory version over the time of the Comanche version. Values greater than one favor Comanche while values less than one favor VMM. From Table 4.1, we can see that the *WaveFront* test cases run faster under Comanche than under VMM.

| Operating System | Data Set size (MB) | Virtual Memory (Second) | Comanche (Second) | Ratio (V/C) |
|---|---|---|---|---|
| Windows98 | 103.68 | 607.6 | 97.2 | 6.25 |
| Window98 | 128 | 792.2 | 130.2 | 5.06 |
| Linux 5.5 | 103.68 | 512.2 | 364.8 | 1.40 |

Table 4.1 WaveFront out-of-core experiments on Windows98 and Linux 5.5

### 4.2 Multitasking

Running the initial test demonstrated an important performance problem with virtual memory's mapping of files. When the first out-of-core tests were run, the windows system became very sluggish and it took a long time for applications to respond. The longer the execution time of the memory mapped code, the worse the problem became. When the Comanche tests were run, this problem did not manifest itself.

A test was constructed to analyze this behavior. Two applications were executed simultaneously (see Table 4.2). The Ratio of the VMM to Comanche execution performance almost doubled between the single out-of-core test and the multitasking out-of-core test. This means that the system performance degrades much more rapidly for VMM than Comanche as more out-of-core applications are executed.

| Operating System | Data Set size (MB) | Virtual Memory (Second) | Comanche (Second) | Ratio (V/C) |
|---|---|---|---|---|
| Windows98 | 103.68 | 1191.3 | 108.2 | 11.7 |
| Window98 | 128 | 1522.9 | 149.9 | 10.16 |
| Linux 5.5 | 32 | 301.1 | 43.6 | 6.91 |

Table 4.2 WaveFront multitasking on Windows98 and Linux 5.5

## 5 Conclusion

The difficulty of handling out-of-core data efficiently limits I/O system performance. Coding out-of-core versions of problems can be a very tedious task and virtual memory system often does not perform well in scientific problems. We believe that there is a need for a need for a compiler-directed explicit I/O approach for regular out-of-core problems.

Our main issue with VMM is its lack of information about the application's actual memory needs and access patterns. As the application continues to fault, the VMM system will start thrashing. Comanche used the source code to determine the maximum number of data sets it needs at any moment n time; this becomes the working set for the

application. Numerous data access patterns have been studied and solutions have been found [3] [14] [16] [17] [18] [19].

The Comanche prototype is sufficient as a proof of concept but it is not quite ready for commercial use. Future enhancements include the design of a user-friendly interface. This allows the extension of semantics of the programming model without the difficulty of extending the syntax of a specific programming language. It also allows multiple views of the same program with different features highlighted.

Another enhancement would be using mathematical expressions to represent the data access patterns and assumptions of each access pattern. The example in this paper is a rather crude model of certain access patterns and the optimization method is presented only as basic ideas. The ability to determine which optimization methods can be chosen and how to use it based on the mathematical expression defining its access pattern is an obvious goal.

## References


[1] Carr, S., McKenley, K. and Tseng, C.-W. *Compiler Optimizations for Improving Data Locality*. Proceedings of 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI), San Jose, CA, October 1994.

[2] Corbert, P., Fietelson, D., Fineberg, S., Hsu, Y. Nitsberg, B., Prost, J., Snir, M., Traversat, B. and Wong, P. *Overview of the MPI-IO Parallel I/O Interface*. Proceedings of 3rd Workshop on I/O in Parallel and Distributed System, IPPS'95, Santa Barbara, CA, April, 1995.

[3] Feng, X. *I/O Performance Improvement through the Use of Compiler-Driven Memory Management*, Master Thesis, Department of Houston, University of Houston, May 2000.

[4] Ferrari, D., Pasquale, J.C., and Polyzos, G.C. *Network Issues for Sequoia 2000.* Proceedings of IEEE Compcon Spring '92, San Francisco, CA, February 1992}.

[5] Han, H., Rivera, G. and Tseng, C.-W. *Compiler and Run-time Support for Improving Locality in Scientific Codes (Extended Abstract).* Proceedings of Languages and Compilers for Parallel Computing, Twelfth International Workshop, Lecture Notes in Computer Science, Springer-Verlag, 1999.

[6] Kandemir, M., Choudhary, A., Ramanujam, J. and Bordawekar, R. *Optimizing out-of-Core Computations in Uniprocessors*. Proceedings of Workshop on Architecture-Compiler Interaction, 3rd HPCA, San Antonio, TX, February 1997.

[7] Kandemir, M., Choudhary, A., Ramanujam, J. and Kandaswamy, M. *A Unified Compiler Algorithm for Optimizing Locality, Parallelism and Communication in Out-of-Core Computations.* Proceedings of IOPADS'97: Workshop on I/O in Parallel and Distributed Systems, San Jose, CA, November 1997 pp. 79-92.

[8] Kandemir, M., Choudhary, A. and Ramanujam, J. *Improving Locality in Out-of-core Computations Using Data Layout Transformations.* Proceedings of the 4th Workshop on Compilers, and Run-Time Systems for Scalable Computers, Pittsburgh, PA, May 1998. Lecture Notes in Computer Science, Vol. 1511, Springer, 1998, pp. 359-366.

[9] Kandemir, M., Choudhary, A., Ramanujam, J. and Bordawekar, R. *Compilation Techniques for Out-of-Core Parallel Computations. Parallel Computing,* 24(3-4), June 1998, pp. 597-628.

[10] Kotz, D. *Multiprocessor File System Interfaces.* Proceedings of the 2$^{nd}$ International Conference on Parallel and Distributed Information Systems, 1993, pp. 194-201.

[11] Leiss, E. L. *Parallel and Vector Computing: A Practical Introduction*. McGraw-Hill, Inc. New York, 1995.

[12] *The Scalable I/O Low-level API: A Portable Programming Interface for Parallel File Systems.* Presentation in Supercomputing'96, Philadelphia, PA, 1996.

[13] *High Performance Computing and Communications: Grand Challenges 1993 Report*. A Report by the Committee of Physics, Mathematical and Engineering Sciences, Federal Coordinating Council for Science, Engineering and Technology.

[14] Robinson, E. M. *Compiler Driven I/O Management*. Ph.D. Dissertation, Department of Computer Science, University of Houston, 1998.

[15] Tseng, C.-W., Anderson, J., Martonosi, M. and Hall, M. *Unified Compilation Techniques for Shared and Distributed Address Space Machines.* Proceedings of 1995 International Conference on Supercomputing (ICS'93), Barcelona, Spain, July 1995.

[16] Zhang, W. and Leiss, E. L. *Block Mapping - A Compiler Driven I/O Management Study*. Proceedings of the 2000 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2000), Las Vegas, Nevada, USA, June 26-29, 2000, pp. 1207-1214.

[17] Zhang, W. and Leiss, E. L. *A Compiler Driven Out-of-Core Programming Approach for Optimizing Data Locality in Loop Nests.* Proceedings of the 2001 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2001), Las Vegas, Nevada, USA, June 25-28, 2001.

[18] Zhang, W *Compiler-Driven I/O Minimization*. Ph.D. Dissertation, Department of Computer Science, University of Houston, August 2001.

[19] Zhang, W. and Leiss, E. L. *Compiler-Driven Runtime I/O Minimization.* Proceedings of the XXVII Conferencia Latinoamericana de Information, Merida, Venezuela, September 24-27, 2001.