Comanche – A Compiler-Driven I/O Management System

Wendy Zhang, Ernst L. Leiss, and Huilin Ye

Abstract—Most scientific programs have large input and output data sets that require out-of-core programming or use virtual memory management (VMM). Out-of-core programming is very error-prone and tedious; as a result, it is generally avoided. However, in many instances VMM is not an effective approach because it often results in substantial performance reduction. In contrast, compiler driven I/O management will allow a program’s data sets to be retrieved in parts, called blocks or tiles. Comanche (COmpiler MANaged caCHE) is a compiler combined with a user level runtime system that can be used to replace standard VMM for out-of-core programs. We describe Comanche and demonstrate on a number of representative problems that it substantially out-performs VMM. Significantly our system does not require any special services from the operating system and does not require modification of the operating system kernel.

Keywords—I/O Management, Out-of-core, Compiler, Tile mapping.

I. INTRODUCTION

The speed gap between processors and disks continues to grow wider with the rapid increase in the performance of processors and communication networks in the last two decades. 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.

The size and complexity of applications, both in the scientific and the commercial world, have increased. While the amount of memory available to high-end programs has certainly increased over the past decades, it is a truism that there is never enough memory. As a consequence, programmers must frequently cope with situations where only portions of the data sets can reside in main memory at any one time while the rest is on disks. As the data sets required by those applications exceed the capacity of the main memory, the computation becomes an out-of-core computation. For out-of-core applications, such as computational fluid dynamics and seismic data processing, which involve a large volume of data, the task of efficiently using the I/O subsystem becomes extremely important. Processing out-of-core data requires staging data in smaller granules that can be fit in the main memory. Data required for the entire computation have to be fetched from files on disk.

Modern computers, including parallel computers, use a sophisticated memory hierarchy consisting of caches, main memory, and disk arrays, to narrow the gap between processor and memory system performance. 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 [2][3][4]. 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 [5][6][7] 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 [8].

Historically, this lack of memory has led to the introduction of virtual memory management (VMM). OS designers offer the handling of I/O activity via VMM. However, VMM typically is quite oblivious to the peculiarities of a specific program and will accommodate “unthinkingly” whatever requests for the retrieval of pages or blocks of memory are passed on to it. This lack of information about the behavior of the executing program can result in very serious inefficiencies [9]. Research in VMM 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.

In the last decade, a number of run-time libraries for out-of-core computations and a few file interfaces have been proposed. SIO [10] proposes a parallel file system-programming interface, which is based on the separation of programmer convenience functions from high performance enabling functions. MPI-IO [11] attempts to provide a portable interface for parallel machines. MPI-IO expresses data partitions in terms of derived data types. The MPI-2 standard [12] has proposed a new, promising interface for parallel I/O. Kotz [13] extends the traditional UNIX file I/O interface for handling the parallel accesses to parallel disk subsystems. 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.

Current operating systems offer poor performance when a numeric application's working set does not fit in main memory. As a result, programmers who wish to solve "out-of-core" problems efficiently are typically faced with the onerous task of rewriting an application to use explicit I/O operations (e.g., read/write) [14]. 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 [15] [16] [17] [18] [19] [20][21][22][23].

In this paper, we concentrate on the compiler-based approach to the I/O problems. Compiler-driven I/O management uses the information collected by a typical optimizing compiler and applies valid code transformations to the program, with the goal of minimizing the number of blocks transferred between disk and main memory. 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 can and 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 [24][25]. It is important to understand that this approach holds great promise for a dramatic improvement in the overall execution time of programs involving fairly regular computation structures, as they are encountered in typical scientific computations. While the techniques involved are also applicable to the transfer of blocks of data between main memory and cache, our interest is more in supplanting VMM, since the access characteristics of main memory of a few nano-second (ns) and disk of tens of milli-second (ms) imply a factor of improvement of seven orders of magnitude while the cache scenario has a factor of improvement of less than 10 (i.e., one order of magnitude).

The ideal array is a square matrix. The Comanche runtime accesses. Data are assumed to be in row major layout on disk. The cooperation between compiler and runtime system is viewed as the cache. In Comanche, the program is restructured to move data automatically between main memory and disk. Most importantly, the compiler is responsible for directing the runtime system as to which data can be transferred and when the data can be transferred. Comanche is written in C and uses the standard C library calls – fopen, fclose, fread, fwrite, and fseek. The runtime system should compile under any compiler that supports these calls. This solution is general purpose and platform independent. The current Comanche system is running under the RedHat 5.0 Linux operating system and Windows 2000 on a PC with a PentiumPro (160Mhz or faster) processor and 64MB or more RAM.

The goal of Comanche is that the system performs at least as well as VMM in all cases and significantly better in most cases. The cooperation between compiler and runtime system is a key factor, as is data dependence analysis, in locality improving optimization. Our research aims at improving and extending the applicability of the Comanche runtime system supported application program interface (API), particularly, the data mapping methodology and optimizing paradigm to achieve high performance of I/O-intensive out-of-core computations.

It is important to keep in mind that in contract to the OS, the compiler has unique information about the data needs of the program, the size and shape of arrays, and the access patterns of the program. It is this information that typically is not available to the operating system, which is customarily charged with the management of I/O calls. Several minor custom optimizations were necessary to achieve across the broad acceptable performance for the programs in our test suite. However, optimizations are invariably better understood when they are applied to specific performance problems.

The standard entity in the Comanche runtime 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. The Comanche runtime system supports a simple application program interface (API). There are four major operations in the API: comanche_map, comanche_unmap, attach, and release. comanche_map takes the file name, the number of rows, the number of columns, the element size in bytes, a flag indicating
if the array is written and the maximum number of rows the application will need to be resident at any given instance as its arguments. It returns a handle to the data structure that represents the array. This call manages the initialization of the data structures needed to support the given array. It is called before the first attach operation.

**comanche_unmap** takes as its single argument the handle returned in the **comanche_map** call. It does not return any value. This call will flush buffers and write everything back to disk, if the flag for array write is set, before returning. It also closes the files opened in **comanche_map**. This call must be paired with a specific **comanche_map** for correct operation and is called after the final release operation.

**comanche_attach** takes as its arguments the handle returned in **comanche_map**, the index of the row to be attached, and a write flag indicating that the row will be subsequently written. It returns a pointer to the beginning of the row. This command will always return the same pointer if repeated calls are made with the same handle and row index. If the row is already in memory, a reference counter is incremented. Otherwise, the row is read from disk. If necessary, it will displace another row whose reference count is zero.

**comanche_release** takes as its arguments the handle returned in **comanche_map** and the index of the row to be released. It does not return a value. This operation signals to the runtime system that a previous call to **comanche_attach** is released and, if the reference count is zero, the space can be reclaimed. It must be paired with a specific **comanche_attach** for correct execution of the system.

The **attach** and **release** operations tell the runtime that a certain region is to be mapped into memory and that an address to the cached data is to be returned. Unlike virtual memory, or cache, the runtime system will not reclaim memory that has been attached no matter how long it has been since it was last referenced. The release operation is enforced by the runtime system and the compiler manages the duration of mapped data and ensures that the number of **attach** operations will not over-fill the available memory before subsequent **release**. Simple reference counting can be used to keep track of how many outstanding **release** operations are left. The **attach** and **release** directives combined with the controlled access to memory allow aliases to be controlled. The cooperation between the compiler and the runtime system achieves a simple but efficient solution.

---

**III. RESULTS ACHIEVED USING COMANCHE**

**A. Most Elements of One Row Are Used Continuously [20], [24]**

The main technique used in Comanche is the use of **attach** and **release** directives. These instructions tell the runtime system that the region is to be mapped into memory and then an address to the called data is to be returned. Each time, **attach** reads a whole row into the buffer in the main memory. The runtime system will not reclaim the memory that has been attached until it is explicitly released, no matter how long it has been since it was last referenced. The compiler manages the duration of mapped data and ensures the data locality in memory. The rationale behind this methodology is the assumption that the neighbor elements are known to be referenced in the future. This method works well when most elements of one row are used continuously and only elements in a few rows are involved each time.

The test suite provides a large coverage of the access patterns common to scientific programs. **stats** uses a one-dimensional vector with sequential read access such as would be found in a statistics collection program. **window1** uses a one-dimensional vector with a window of neighbors similar to a finite difference computation in one dimension. **matvec** is a standard matrix vector product, \( C = A \times B \). The compiler manages the resource and accesses three two-dimensional grids in a **stencil** pattern like that found in the 2D wave equation. The system performs better than VMN and uses significantly fewer system resources to do so.

The test suite was executed on a standard Unix workstation. The workstation is a PentiumPro 200MHz processor with 64MB of RAM running the Linux operating system. Tests were performed on an unloaded system without the X-windows server; only the applications and basic system service were running. Problem parameters were adjusted to use 80MB of data. Codes were tested first by comparing output on several input test cases. Table 1 shows the execution times.

**TABLE I**

<table>
<thead>
<tr>
<th></th>
<th>Virtual Memory</th>
<th>Comanche</th>
<th>Ratio (V/C)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>stats</strong></td>
<td>4:58</td>
<td>1:52</td>
<td>2.66</td>
</tr>
<tr>
<td><strong>window1</strong></td>
<td>11:51</td>
<td>4:41</td>
<td>2.53</td>
</tr>
<tr>
<td><strong>stencil</strong></td>
<td>33:49</td>
<td>25:17</td>
<td>1:34</td>
</tr>
<tr>
<td><strong>matvec</strong></td>
<td>4:33</td>
<td>2:16</td>
<td>2.01</td>
</tr>
</tbody>
</table>

A test was constructed to compare multitasking execution times. Two applications were executed simultaneously on four data files (see Table 2). The ratio of the VMM to Comanche execution performance more than doubled between the single out-of-core test and the multitasking out-of-core test. This means that system performance degrades much more rapidly for VMM than Comanche as more and more out-of-core applications are executed.

**TABLE II**

<table>
<thead>
<tr>
<th></th>
<th>Virtual Memory</th>
<th>Comanche</th>
<th>Ratio (V/C)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>matvec</strong></td>
<td>1:26:18</td>
<td>15:40</td>
<td>5.51</td>
</tr>
<tr>
<td>1:23:53</td>
<td>15:04</td>
<td>5.57</td>
<td></td>
</tr>
<tr>
<td>1:22:37</td>
<td>15:00</td>
<td>5.51</td>
<td></td>
</tr>
<tr>
<td>1:26:04</td>
<td>15:41</td>
<td>5.49</td>
<td></td>
</tr>
<tr>
<td><strong>stats</strong></td>
<td>1:26:14</td>
<td>15:48</td>
<td>5.46</td>
</tr>
<tr>
<td>1:23:46</td>
<td>15:17</td>
<td>5.41</td>
<td></td>
</tr>
<tr>
<td>1:23:15</td>
<td>15:18</td>
<td>5.41</td>
<td></td>
</tr>
<tr>
<td>1:26:00</td>
<td>15:47</td>
<td>5.49</td>
<td></td>
</tr>
</tbody>
</table>

**B. Elements in Many Rows Are Involved [26], [27]**

The original Comanche data structures are modified to allow a section of a row to be attached to the buffer instead of the whole row. A simple optimization based on a **seeker/reaper** paradigm is used. **wavefront** stands for an \( N \times N \) square with tilted lines inside the array \( A \) in waves \( W_i \) through \( W_{2n-1} \). **rhombus** considers a square in which there are many equal-sized “tall and lean” rhombuses. **zigzag** runs across every row.
The results reported in these three experiments, *wavefront*<sup>rhombus</sup> and *zigzag*, demonstrate that the modified Comanche API can achieve better I/O performance in cases where at each time, elements in many rows are involved and for each row, only a few elements are needed.

### B.1 wavefront

*WaveFront* stands for a square with tilted lines inside, the array A in waves W<sub>1</sub> through W<sub>2n-1</sub> as follows (assume N = 4):

- W<sub>1</sub>: A(1,1)
- W<sub>2</sub>: A(2,1), A(1,2)
- W<sub>3</sub>: A(3,1), A(2,2), A(1,3)
- W<sub>4</sub>: A(4,1), A(3,2), A(2,3), A(1,4)
- W<sub>5</sub>: A(4,2), A(3,3), A(2,4)
- W<sub>6</sub>: A(4,3), A(3,4)
- W<sub>n</sub>: A(N,1)

In general, we have:

- W<sub>i</sub>: A(i, 1), A(i-1, 2), …, A(2, i-1), A(1, i) for i = 1, …, N
- W<sub>N+j</sub>: A(N, j+1), A(N-1, j+2), …, A(j+2, N-1), A(j+1, N) for j = 1, …, N-1

We have run the algorithm as an out-of-core program written in the C language as produced by Comanche. The C compiler generates working code using the Comanche runtime system. The resulting code was executed 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.

One set of experiments used 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).

Initial test runs demonstrated an important performance problem with the virtual memory 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. In the Comanche tests this problem did not manifest itself.

A test was constructed to analyze this behavior. Two applications were executed simultaneously (see Table 3). 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 for Comanche as more out-of-core applications are executed.

### B.2 rhombus

This task traverses each rhombus, calculates the sum of all the elements on it, and writes the total value to a file. We assume that the length of the side of the square is many times larger than the rhombus’s width, in other words, each rhombus strides across all the rows, but straddles a very limited number of columns. We will begin from the element on the apex of the first rhombus and proceed toward its next row until we reach the last row in the square. During this process, it adds up all the elements that happen to be on the rhombus. After finishing traversing the first rhombus, we need to go back to the first row of the square again. We will choose the next element, which is the apex of the second rhombus and repeat the whole traversing process. We keep traversing every rhombus in the square until we finish the rightmost one. An important observation is that in the traversal we need almost every row even though we may need just one or two elements of each row and we need go back and reference some other elements of those rows again after finishing the previous traversal.

We applied rotation to the optimization of the rhombus problem. We allocated two groups of subrows to the buffers, and we assumed that the width of the rhombus is less than the length of the subrow. Our purpose is that at any time the elements on each rhombus must be in the buffers. For each row on which rhombuses have elements, there are two groups of subrows. They keep moving forward toward the next column while rotating their positions. Whenever the rhombus is to move out of subrow #1 into subrow #2, subrow #1 will be shifted right next to subrow #2; similarly for those subrows on the next rows, such as #3, #4, #5, #6, and so on. Every row on which rhombuses have elements repeats the same process. This movement gives the illusion that each rhombus is in the buffers all the time since the buffers are moving along as the rhombus references are moving. Buffer movement is always one step ahead of the rhombus reference movement, which guarantees that buffers always have the ability to provide the data that are soon to be referenced. At the end of the program, all subrows in buffers are released. We have run one set of experiments for a double precision matrix of size 3600 x 3600 involved in the Rhombus application. The total data set space is 3600 x 3600 x 8 = 103MB. Data files were initialized with random values in the in the interval (-1, +1). The code is omitted due to limited space (see [26]). Two applications were executed simultaneously for the multitasking case. 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 1 favor Comanche while values less than 1 favor VMM. From Table 4, it is obvious that the system performance is optimizes dramatically under Comanche.

### Table III

<table>
<thead>
<tr>
<th>Operating System</th>
<th>Data Set size (MB)</th>
<th>Virtual Memory (second)</th>
<th>Comanche (second)</th>
<th>Ratio (V/C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows98</td>
<td>103.68</td>
<td>1191.3</td>
<td>108.2</td>
<td>11.7</td>
</tr>
<tr>
<td>Window98</td>
<td>128</td>
<td>1522.9</td>
<td>149.9</td>
<td>10.16</td>
</tr>
<tr>
<td>Linux 5.5</td>
<td>32</td>
<td>301.1</td>
<td>43.6</td>
<td>6.91</td>
</tr>
</tbody>
</table>

### Table IV

<table>
<thead>
<tr>
<th>Execution Type</th>
<th>Data Set Size (MB)</th>
<th>Virtual Memory (second)</th>
<th>Comanche (second)</th>
<th>Ratio (V/C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Out-of-core</td>
<td>103</td>
<td>287.4</td>
<td>28.6</td>
<td>10.5</td>
</tr>
<tr>
<td>Multitasked</td>
<td>103</td>
<td>619.1</td>
<td>28.9</td>
<td>21.42</td>
</tr>
</tbody>
</table>
B.3 zigzags

Instead of a rhombus, we have zigzags in a square. e traverse each zigzag, calculate the sum of the elements on the zigzag, and write the total to a file. The main difference between the rhombus case and the zigzag case is that rhombus may skip over some rows, but zigzag does not. Since rhombus has four tilted lines as its four sides, these lines may not always intersect a row at an element, and they could pass through the space between two neighbor elements. On the other hand, zigzag runs across every row. It always has elements on each row, even though the number of elements on each row may be different: sometimes the number is equal to the width of the zigzag, but for most rows, the number is 1.

As in the rhombus test case, we applied rotation in our optimization. We allocated two groups of subrows in the buffers; they can rotate forward toward the next column. We assumed that the width of the zigzag is less than the length of the subrows. Buffer movement is always one step ahead of the zigzag reference movement.

We have run one set of experiments for a double precision matrix of size 4000 x 4000 involved in the zigzags application. The total data set space is 4000 x 4000 x 8 = 125MB. Data files were initialized with random values in the in the interval (-1, +1). The code is omitted due to limited space (see [26]). From the experimental data in Table 5, we can see that the modified Comanche system outperforms VMM for this zigzag access pattern.

<table>
<thead>
<tr>
<th>Table V</th>
<th>ZIGZAG OUT-OF-CORE AND MULTITASKING EXPERIMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution Type</td>
<td>Data Set Size (MB)</td>
</tr>
<tr>
<td>Out-of-core</td>
<td>125</td>
</tr>
<tr>
<td>Multitasked</td>
<td>125</td>
</tr>
</tbody>
</table>

C. Matrices Involving Crossing Terms [24], [25], [28], [30]

Matrix transpose and matrix multiplication have a crisscross access pattern. Traversal along one dimension implies simultaneous traversal along another dimension. When calculations on matrices involve crossing index terms, a common solution is blocking (tiling) [32]. Tiling plus a good process scheme may provide significant improvement.

C.1 Matrix Transpose

Matrix transpose (matran) involves cross index terms so without restructuring terrible performance is to be expected. Tiling provides an improvement to both the virtual memory version and the Comanche version. Space is allocated for two submatrices or tiles of the original matrix. Transposition is performed on tiles that represent the transpose of each order. The transposition can be done in-core and then the results are written all at once back to disk.

We tested tiling by constructing the tiled extensions to Comanche; then we tested a special version of matran that was hand coded for the tiled extensions. We compared VMM and Comanche two systems for the volume of data transferred. The ratio of the total bytes transferred in the Comanche simulation to the total bytes transferred in the virtual memory simulation is reported in Table 6. The ratio of total bytes transferred has dropped to well below 1 which implies that the VMM matrix transpose code is not efficient; this is due of course to the fact that it has not been tiled. This illustrates the importance of data dependence analysis in making compiler driven I/O management feasible. There is a 700 fold improvement in performance, nearly three orders of magnitude.

<table>
<thead>
<tr>
<th>Table VI</th>
<th>Matrix Transpose Before and After Optimization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ratio</td>
<td>Before</td>
</tr>
<tr>
<td>bw</td>
<td>lat</td>
</tr>
<tr>
<td>0.9</td>
<td>121.12</td>
</tr>
<tr>
<td>0.8</td>
<td>73.43</td>
</tr>
<tr>
<td>0.7</td>
<td>72.12</td>
</tr>
<tr>
<td>0.6</td>
<td>74.27</td>
</tr>
<tr>
<td>0.5</td>
<td>78.59</td>
</tr>
</tbody>
</table>

C.1.1 Matrices Involving Crossing Terms

Assume there are three matrices A, B, and C of size (N, N) on disk. We will multiply A by B and store the result in the matrix C. To store the matrices A, B, and C in memory requires 3N^2 space. If N is large, the total amount of main memory available is less than the needed space of 3N^2. If the data cannot be entirely loaded into memory, the problem becomes out-of-core.

The traditional way of coding this in C is something like this:

\[
\begin{align*}
&\text{for } (i = 0; i < N; i++) \\
&\quad\text{for } (j = 0; j < N; j++) \\
&\quad\quad\text{for } (k = 0; k < N; k++) \\
&\quad\quad\quad C[i,j] += A[i,k] * B[k,j];
\end{align*}
\]

Note that although the same row of A is reused in the next iteration of the middle loop, a large volume of data used in the intervening iterations may be replaced. During processing, a virtual memory system would do much swapping, which is very time consuming.

One solution to solve this out-of-core problem is to split each matrix into several sub-matrices (blocks) of size (M, M). Specially, if the dimensions of the matrix are divisible by the dimensions of the block we can use an algorithm suggested by the following observations.

Assume M is one half of N; then each matrix can be split into four sub-matrices of size (N/2, N/2). Schematically we have:

\[
\begin{align*}
A_{11} &\quad A_{12} &\quad x_{B_{11}} &\quad B_{12} = C_{11} &\quad C_{12} \\
A_{21} &\quad A_{22} &\quad x_{B_{21}} &\quad B_{22} = C_{21} &\quad C_{22}
\end{align*}
\]

The C_{ij}’s are defined as follows:

\[
\begin{align*}
C_{11} &= A_{11} \cdot B_{11} + A_{12} \cdot B_{21} \\
C_{12} &= A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\
C_{21} &= A_{21} \cdot B_{11} + A_{22} \cdot B_{21} \\
C_{22} &= A_{21} \cdot B_{12} + A_{22} \cdot B_{22}
\end{align*}
\]

Assuming all C_{ij} are initialized to zero, the following instructions can be executed in any order:

\[
C_{11} = C_{11} + A_{11} \cdot B_{11}
\]
\[ C_{11} = C_{11} + A_{12} \cdot B_{21} \]
\[ C_{12} = C_{12} + A_{11} \cdot B_{12} \]
\[ C_{12} = C_{12} + A_{12} \cdot B_{22} \]
\[ C_{21} = C_{21} + A_{21} \cdot B_{11} \]
\[ C_{21} = C_{21} + A_{22} \cdot B_{21} \]
\[ C_{22} = C_{22} + A_{21} \cdot B_{12} \]
\[ C_{22} = C_{22} + A_{22} \cdot B_{22} \]

A brute force approach might retrieve each of the twenty-four sub-matrices (blocks) regardless of reusability.

In out-of-core programming, we can sequence the instructions in order to reuse sub-matrices as often as possible. We use the Comanche runtime function block_attach when we need a block and block_release when we no longer need that block.

Assume that the amount of available main memory is equal to four sub-matrix, \(4 \cdot N^2/4\). This means that we are able to keep a whole row of blocks of the A matrix, one block of the B matrix, and one block of the matrix C in memory during processing. The following sketches an efficient approach (see [1], [32]):

\[ \text{C11 A11 B11} \]
\[ \text{C11 B21 A12} \]
\[ \text{C12 A11 B12} \]
\[ \text{C12 B21 A12} \]
\[ \text{C21 A21 B11} \]
\[ \text{C21 B21 A22} \]
\[ \text{C22 A21 B12} \]
\[ \text{C22 B22 A22} \]
\[ \text{Release all} \]

In this scheme, we only retrieve eight blocks from disk and store four blocks to disk. We use a similar, but more complicated scheme if \( N \) is not divisible by \( M \).

We have implemented the general algorithm in an out-of-core program written in the C language. The C compiler under Linux generates working code using the Comanche runtime system. On a single processor PC with a PentiumII, 333MHz microprocessor, the code was run under Redhat Linux 5.0 in command mode. The system used for the actual performance testing was a 3GHz Intel Xeon, 64MB of memory of which about 50MB are available to the program.

We have run one set of experiments for multiplying 1024 x 1024 double precision matrices. The total data set space is 1024 x 1024 x 8 x 3 = 24MB. Another set of experiments was run for 2048 x 2048 matrices, with a total data set of 96MB. Data files are initialized with random values in the interval \((-1, +1)\).

We have measured the performance of the matrix multiplication for different block sizes, namely 512 x 512, 256 x 256, 128 x 128, 64 x 64, 32 x 32, 16 x 16, and 8 x 8. We applied these different block sizes to the data sets of 24MB and 96MB. The performance values are listed in Tables 7 and 8.

We observe that block multiplication uses only 24.5% to 55.0% of the execution time of regular multiplication (VMM in Table 7). The one exception is for a laughably small block size (8 x 8).

### D. Loop-Carried Dependences [25], [29], [30]

Calculations on matrices often involve loop-carried dependences on columns, rows, and strides. Scientific computations such as Fast Fourier Transform (FFT) and Multi-grid use increasing and decreasing strides. Stepped programs traverse a two-dimensional grid using step sizes that vary by powers of two. This program imposes an important limitation on the simpler, row base API, as the stride grows beyond the size of a single page; reading an entire row will consist mostly of dead weight [24]. A naively implemented stepped program running under Comanche is considerably slower than under VMM, in real test cases [24]. The difficulty with this access pattern is that it is not just crisscross but also overlapped. When two adjacent loops have the same loop limits, they can sometimes be fused into a single loop. Tiling, subarray, loop fusing, the seeker/reaper paradigm, plus a good process scheme provide significant improvement.

### TABLE VII

<table>
<thead>
<tr>
<th>Matrix size</th>
<th>Elapsed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024 x 1024</td>
<td>31:05.7</td>
</tr>
<tr>
<td>2048 x 2048</td>
<td>4:31:27</td>
</tr>
</tbody>
</table>

### TABLE VIII

<table>
<thead>
<tr>
<th>Block Size</th>
<th>Elapsed</th>
</tr>
</thead>
<tbody>
<tr>
<td>512 x 512</td>
<td>09:25.0</td>
</tr>
<tr>
<td>256 x 256</td>
<td>07:30.0</td>
</tr>
<tr>
<td>128 x 128</td>
<td>07:59.0</td>
</tr>
<tr>
<td>64 x 64</td>
<td>07:46.0</td>
</tr>
<tr>
<td>32 x 32</td>
<td>13:50.0</td>
</tr>
<tr>
<td>16 x 16</td>
<td>15:30.0</td>
</tr>
<tr>
<td>8 x 8</td>
<td>42:33.0</td>
</tr>
</tbody>
</table>

| # of Seek  | 40,960  |
| # of Read  | 30720   |
| # of Write | 10,240  |
| Memory usage | 10,240 |

### TABLE IX

<table>
<thead>
<tr>
<th>Block Size</th>
<th>Elapsed</th>
</tr>
</thead>
<tbody>
<tr>
<td>512 x 512</td>
<td>14:00.8</td>
</tr>
<tr>
<td>256 x 256</td>
<td>14:09.0</td>
</tr>
<tr>
<td>128 x 128</td>
<td>13:32.0</td>
</tr>
<tr>
<td>64 x 64</td>
<td>11:01.0</td>
</tr>
<tr>
<td>32 x 32</td>
<td>1:02:11</td>
</tr>
<tr>
<td>16 x 16</td>
<td>6:41:08</td>
</tr>
<tr>
<td>8 x 8</td>
<td>6:41:08</td>
</tr>
</tbody>
</table>

| # of Seek  | 245,760 |
| # of Read  | 204,800 |
| # of Write | 40,960  |
| Memory usage | 12,583,383 |
Since our program is out-of-core, we cannot load the whole array into memory. We tile the array into several super-rows (blocks). In each tile, we seek and attach two rows at a time. First, we attach two rows to perform the inner loop (column control) of step size 1. Then we have to fuse a stepped loop with row and column loop control to seek another row to compute. When we fuse the stepped loop into the row loop, we have to keep the original reference order. There is a loop dependence, for instance, before we can perform the step size 2 calculations on rows 0 and 2, since both rows must have already completed the step size 1 calculation.

Because of the loop dependence, we have to seek half a block plus 1 element ahead for each block. During the calculation, we release any row which is no longer needed in future calculations. After finishing the calculation on one super-row, we shrink the first row of the block into a sub-array for further calculation. We perform stepped calculation on the sub-array starting with step size 1 and return the final results to the original rows. We use the reaper to release all the rows and terminate the calculation. We have added to Comanche a subarray data structure:

To illustrate this process, assume we have a matrix of size 12 x 12 and a block size of 4; the calculation is denoted as \( \otimes \). Here is the complete program; we assume that \( R_i \) denotes row \( i, i = 0, 1, \ldots, 11 \):

1. seek \( R_0 \), attach it;
2. attach \( R_1 \) and \( R_2 \);
3. \( R_0 \otimes R_1 \) step size 1, \( R_1 \otimes R_2 \) step size 1;
4. release \( R_1 \);
5. attach \( R_3 \) and \( R_4 \);
6. \( R_2 \otimes R_3 \) step size 1, \( R_3 \otimes R_4 \) step size 1;
7. release \( R_3 \);
8. \( R_2 \otimes R_2 \) step size 2;
9. attach \( R_5 \) and \( R_6 \);
10. \( R_5 \otimes R_5 \) step size 1, \( R_3 \otimes R_6 \) step size 1;
11. release \( R_5 \);
12. \( R_2 \otimes R_4 \) step size 2;
13. release \( R_2 \);
14. put \( R_4 \) into subarray;
15. attach \( R_3 \) and \( R_4 \);
16. \( R_6 \otimes R_7 \) step size 1, \( R_7 \otimes R_8 \) step size 1;
17. release \( R_7 \);
18. \( R_3 \otimes R_6 \) step size 2;
19. attach \( R_0 \) and \( R_6 \);
20. \( R_8 \otimes R_9 \) step size 1, \( R_9 \otimes R_{10} \) step size 1;
21. release \( R_9 \);
22. \( R_6 \otimes R_8 \) step size 2;
23. release \( R_6 \);
24. put \( R_8 \) into subarray;
25. attach \( R_{11} \);
26. \( R_{10} \otimes R_{11} \) step size 1;
27. release \( R_{11} \);
28. \( R_8 \otimes R_{10} \) step size 2;
29. release \( R_{10} \);
30. put \( R_8 \) into subarray
31. inside subarray \( R_0 \otimes R_4 \) step size 1;
32. inside subarray \( R_2 \otimes R_8 \) step size 1;
33. inside subarray \( R_2 \otimes R_8 \) step size 2;
34. write back to \( R_0 \) and release \( R_0 \);
35. write back to \( R_2 \) and release \( R_4 \);
36. write back to \( R_8 \) and release \( R_8 \).

Note that we can change the execution order if this does not violate the data dependencies. This schema performs tiled stepped calculations without subarrays.

We have implemented the general algorithm suggested by these comments in an out-of-core program written in the C language. The C compiler under Linux generates working code using the Comanche runtime system. On a single processor PC with a PentiumI, 333MHz microprocessor, the code was run under Redhat Linux 5.0 in command mode. The system used for the actual performance has 64MB of memory, of which about 50MB are available to the program.

We have run the same set of experiments for a double precision matrix of size 3072 x 3072 involved in the stepped calculation. The total data set space is 3072 x 3072 x 8 = 72MB. Another set of experiments was run for the size 6144 x 6144 with a total data set space of 288MB. We also have run a set of experiments for multiple tasks with a data set space of 128 MB. Data files were initialized with random values in the interval (-1, +1).

To implement tiled stepped calculations, we divided the matrix into blocks. Each block has size \( M \). Then we performed the stepped calculation sketched in Section 3.3. We mapped two rows of the matrix \( A \) into memory. After one round of calculation, we released the row which is no longer needed in future calculations. We started over for another row until the whole stepped calculations were finished.

For a matrix of size \( N \times N \), the memory space needed (in bytes) is \( N \times N \times 8 \). If the block size is \( M \), the number of needed buffers, \( B \), is \( \log_2 N + M/2 +3 \) and the size of the subarrays, \( S \), is \( N/M \times N/M \). The memory space needed for our tiled subarray method is \( B \times N + S \). This method works for any block size that is a power of 2 and the matrix size is divisible by the block size.

For comparison, we have measured the performance of the tiled stepped calculation for different block sizes. We ran experiments for the block sizes 512, 256, 128, 64, 32, 16, and 8. We applied these different block sizes to the data sets of size 72MB and 288MB. The performance values are listed in Tables 11 and 12.

From the experiments we see that tiling and subarray stepped calculations use only 21.8% to 49.7% of the execution time of the regular stepped calculation listed in Table 10. The experiments on different block and subarray sizes suggest that the block and subarray sizes do not play an important role in the performance.

### Table X

**Performance of Regular Stepped Calculations**

<table>
<thead>
<tr>
<th>Matrix Size</th>
<th>Elapsed</th>
<th>Page fault</th>
</tr>
</thead>
<tbody>
<tr>
<td>3072 x 3072</td>
<td>07:56.8</td>
<td>181,083</td>
</tr>
<tr>
<td>6144 x 6144</td>
<td>1:06:52</td>
<td>1,320,021</td>
</tr>
</tbody>
</table>
IV. CONCLUSION, LIMITATION, AND FUTURE STUDIES

Comanche is a software system whose goal is to reduce the amount of implicit I/O of a given program. Comanche’s approach is based on information collected by a typical high-performance optimizing compiler, in particular dependence analysis; it attempts to restructure the program using standard code transformation techniques. Comanche was initially developed as a proof of concept and is at present essentially restricted to uniprocessors. It has been used to demonstrate convincingly the superiority of compiler-driven I/O management over VMM on a variety of scientific computation applications.

The Comanche runtime system provides efficient out-of-core programming without suffering from the disadvantages of virtual memory management. The system performs better than VMM in all test cases and uses significantly fewer system resources. We have combined key components on top of established compiler technology to build an application program interface (API). The compiler managed cache approach guides the runtime system in the management of resources. This division of responsibilities greatly simplifies the overall model while still supporting high performance. This approach proved successful as demonstrated by the prototype.

The Comanche prototype is sufficient as a proof that compiler managed I/O of out-of-core computations outperforms VMM. While the concept has been proven and key problems have been solved, several issues remain; therefore it is not quite ready for commercial use.

We plan to reengineer Comanche to adapt it to the high-end computing needs of cutting-edge scientific and engineering work. This will be accomplished in transitioning from a proof-of-concept to a generally usable tool with significant implications for usability, applicability, and reliability.

A. Out-of-Core Parallel Computation

The difficulty of handling out-of-core data limits the performance of parallel machines and distributed systems. I/O management is very important in distributed and parallel computing. The computation on out-of-core data often requires data which are not present in a processor’s memory, requiring I/O access as well as communication. The compiler should consider the individual communication requirements of each processor.

An area of future work is extending our concepts to multiprocessor and distributed computing environments involving many thousands of processors and integrating Comanche with the techniques designed to eliminate I/O costs originating from communication requirements of out-of-core parallel programs.

B. Migration and Data Profiling

Number An initial set of guidelines for migrating legacy codes into the Comanche architecture has been presented. Techniques for recognizing common programmer practices would be instrumental in effective migration of real world programs. In order to become more practical, a significant amount of effort in migration is needed.

An I/O profile describes how much I/O has been performed and where; additionally it will represent information related to I/O wait time. The I/O profile may pinpoint where “too many” data transfers have occurred; therefore it provides guidance in improving the I/O behavior of a program. In this way, the I/O profile can help in predicting the amount of I/O and the run time required for a program similar to the one on which the I/O profile is based. A compile time data transfer analyzer computes an approximation on the number of data transfers between main and secondary memory when the Least Recently Used (LRU) replacement policy is applied. When a program requires too much I/O work, program restructuring to minimize the data transfers becomes necessary. This approximation can be used as guidance to transform I/O-intensive programs to achieve better performance.

We plan to integrate our work of I/O profiling [30] into the next version of Comanche. The analyzer automatically estimates the number of blocks or pages transfers that a program needs for execution, based on the parameters of the
system currently available. The data profile reflects the real behavior of the system when executing the program and can therefore be used to compare situations derived for the same program after restructuring.

We also plan to integrate our work on automatic reduction of memory bank conflicts [33] into the improved version of Comanche. This is of particular concern when dealing with high-performance programming environments where bank conflicts are frequently underestimated and overlooked. A technique for supporting libraries is needed and developing a practical migration tool should be pursued.

REFERENCES