A Compiler Driven Out-of-Core Programming Approach for Optimizing Data Locality in Loop Nests

Wendy Zhang  
Department of Computer Information Systems  
Southern University at New Orleans  
New Orleans, LA, USA

Ernst L. Leiss  
Computer Science Department  
University of Houston  
Houston, TX, USA

Abstract Most scientific programs have large input and output data sets that require out-of-core programming or the use of virtual memory management (VMM). Often, VMM is not an effective approach because it results frequently in substantial performance reduction. In contrast, compiler driven I/O management allows a program's data sets to be retrieved explicitly in parts, called blocks or tiles. In this paper, we offer an out-of-core programming schema to optimize locality of disk accesses in stepped problems by choosing a good combination of data tiling and loop transformations. The experimental results provide strong evidence that an out-of-core programming approach combined with non-standard data mappings and corresponding loop transformations can improve the performance of out-of-core problems by one order of magnitude or more.

Keywords: I/O management, tile mapping, VMM, out-of-core, compiler

1 Introduction

Out-of-core applications, such as scientific computing with massive data sets, databases, image visualization, and multimedia applications, are likely to suffer from a bottleneck between memory and disk. Virtual Memory Management (VMM) system uses a static policy for storage management. As pointed out in [Leiss95], this uniform page automatic transfer method is convenient to use, but programmers are no longer fully in control of the program. Numerous programs with large data sets do not perform well when they rely on virtual memory management. There is a clear need for compiler directed explicit I/O for out-of-core computations [1] [2] [3] [4] [5] [12][14].

Writing an efficient out-of-core version of a program is a difficult task; managing it will require a combination of memory management, out-of-core compilation, and the interaction of file layouts on disks and loop transformations. Comanche [11] is based on a compiler-managed cache where the main memory accommodates the cache. The compiler has unique information about the data access patterns of the program, and the size and shape of arrays. Note that it is this information that typically is not available to the operating system, which is customarily charged with the management of I/O calls. Comanche restricts the out-of-core program to move data more efficiently between main memory and disk. More details about Comanche will be given in Section 2.

We want to maximize the locality of data references in order to reduce I/O transfers between layers of the memory hierarchy. We may take advantage of program locality by focusing our optimization on loops. It is often not very difficult to do I/O optimization for a specific loop or nest of loops. However, it is not easy to do I/O optimization for an entire program. [6] points out instances where a straightforward extension of the traditional tiling method to I/O-intensive programs may result in poor I/O performance due to the particular characteristics of I/O operations.

One way to overcome the problem is to transform the program and the data sets together such that the localities between those two are maintained. Tiling (blocking) [9] is a technique
to improve locality. Tiling of out-of-core data into memory is an out-of-core compilation strategy based on explicit I/O. The main issue is to select the appropriate loop order and mapping method such that the overall I/O time will be minimized. [5] shows that the determination of tile access patterns can be successfully achieved by compiler analysis. [10] demonstrates that the application of compound transformations consisting of loop permutation, loop fusion, loop distribution, and loop reversal is useful for optimizing out-of-core programs. [13][7] describe a locality optimization algorithm that applies a combination of loop interchanges, skewing, reversal, and tiling to improve the data locality of loop nests. [8] describes an I/O minimal algorithm where for a given amount of space the number of retrievals of blocks from external memory is minimal.

Out-of-core stepped problems are used in this paper to study the importance of combining data layout and loop transformations. An I/O minimal algorithm for a stepped problem is sketched which forms the basis of this work. In Section 3, we discuss the stepped problem and the amount of space required to carry out the computation as well as the I/O scheme used with the Comanche system.

In testing the proposed stepped algorithm, it became clear that loop transformation plays a very important role. In Section 4, we will compare the performance for different block sizes. 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 [11]. 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 5.0 Linux operating system on a PC with a PentiumPro (160Mhz or faster) processor and 64MB or more RAM.

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 (block_s) is a block structure used to buffer a block of data.

typedef structure block_s {
    double *data;
    int flag;
    int refcnt;
    int rowindex;
    int columnindex;
    int nrow;
    int ncol;
    int blockno;
} Block;

The other structure is a matrix structure (array_s) that holds the information of the matrix on disk and has buffers to hold several blocks in memory.

typedef struct array_s {
    int nblocks;
    int nelems
    int bsize;
    int elesize;
    FILE *fp;
    long offset;
    char *name;
    int *map;
    int nbuffers;
    Block **buffers;
    int victim;
    int flag;
    int total_mem;
} Matrix;

Besides these structures, there are several functions in Comanche. Two major functions are attach_block and release_block which are used to perform the block 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 blocks. Comanche will take a row of an array and cut it up into blocks, so that a block only corresponds to a single row or a sub-matrix. 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 attach_block and release_block 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 block 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 Stepped Problems

Scientific computations such as Fast Fourier Transforms (FFT) and Multi-grid methods use increasing and decreasing strides. *Stepped* programs traverse a two-dimensional grid using step sizes that are powers of two. Figure 3.1 illustrates this on a portion of the array with step sizes (A) 1, (B) 2, (C) 4, and (D) 8. A naively implemented stepped program running under Comanche is considerably slower than under VMM, in real test cases. Such programs reveal limitations of 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 [11].

![Figure 3.1 Stepped Access Patterns](image)

The original code is as follows:

```c
void stepped(int n, double [][]A) {
  int i, j, k;
  double d0, d1;
  for (k = 1; k < n; k <<= 1) {
    for (i = 0; i < n - k; i += k) {
      for (j = 0; j < n - k; j += k) {
        d0 = A[i][j]/2.0;
        A[i][j] = d0 + d1;
      }
    }
  }
}
```

Since the primary reason for the poor performance in this example is the high amount of dead weight that occurs when the strides become large, we are studying subarray optimization to avoid reading the dead weight in the rows. *Subarrays* represent mappings from remote storage to local storage using non-unit strides. An example of an array starting at 2, and consisting of every fourth element, up to 24, is illustrated in Figure 3.2. In this way, we map an array of size 24 into an array of size 6 to save the memory space and reduce I/O bandwidth. The initial results of our experiments using this naïve approach were disappointing.

![Figure 3.2 Subarrays](image)

When we take a closer look at the stepped program, we realize that the difficulty of this access pattern is that it is not just crisscross but also overlapped. It appears that the performance of the stepped calculation cannot be improved by using data mapping optimization alone. When two adjacent loops have the same loop limits, they can sometimes be *fused* into a single loop. To solve our stepped problem, there is no single loop transformation method or mapping pattern available. As a result, we combined tiling, subarrays, loop fusing, and the seeker/reaper paradigm of Comanche in combination. The ultimate goal of our study is to develop a tool based on Comanche for determining which optimization method can be chosen and how to use it based on the access pattern.

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 perform. 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.

We have added to Comanche a subarray data structure:
typedef struct type_t {
  double *data;
  int nrow;
  int ncol;
  int start;
  int stop;
  int step;
} Subarray;

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.

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

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

### 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 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 has 64MB of memory of that about 50MB are available to the program.

We have run one 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).

The Linux implementation requires that we execute tests on an unloaded machine and use elapsed time. For each experiment, we have recorded the following values:

1. Block size: given as number of rows and columns;
2. Elapsed: time used for executing the whole program;
3. Page fault: number of page faults in the virtual memory system;
4. $\#$ of Seek: the number of seek function calls used in Comanche;
5. $\#$ of Read: the number of read function calls used to map data from disk to memory;
6. # of Write: the number of write function calls used to map data from memory to disk;
7. Memory usage: the memory locations allocated to the data structure used to hold the blocks in memory.

During initial testing we found that block mapping large files has a significant impact on the system resources. We did additional tests to assess the impact of different block sizes on run time.

4.1 Stepped Calculations

In ordinary stepped calculations, the matrix A is mapped into memory by an mmap system call. Virtual memory is tested by mapping files into memory in the conventional way.

The performance values are given in Figure 4.1. Compared with the tiled stepped calculation, this method causes more page faults so the elapsed time is generally large.

4.2 Tiled Stepped Calculations with Subarrays

To implement tiled stepped calculations, we divided the matrix into blocks. Each block has size M. Then we perform the stepped calculation sketched in Section 3. We map two rows of the matrix A into memory. After one round of calculation, we release the row which is no longer needed in future calculations. We start over for another row until the whole stepped calculations are finished.

For a matrix of size N x N, the memory space needed (in bytes) is N x N x 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 x N/M. The memory space needed for our tiled subarray method is B x 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. (The code is not given here due to space limitations).

For comparison, we have measured the performance of the tiled stepped calculation for different block sizes. We did experiments on 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 Figures 4.2 and 4.3.

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. The experiments on different block and subarray sizes suggest that the block and subarray sizes do not play an important role in the performance.
4.3 Tiled Stepped Calculations without Subarrays

To implement tiled stepped calculation, we map two rows of the matrix $A$ into memory. After one round of calculation, we release the row which is no longer needed in future calculations. We start over for another row until the whole stepped calculation is finished. Assuming the size of matrix is $N \times N$, the number of the buffers, $B$, is $\log_2 N + 3$. The memory space needed is $B \times N$. This schema works for any even matrix size. (The code is not given here due to space limitations).

Again, we have measured the performance of stepped calculation for different data sets of size 72MB and 288MB. The performance values are listed in Figure 4.4. From the experiments we see that tiled stepped calculations use only 21.8% to 46.5% of the execution time of the regular stepped calculation.

5 Conclusion

Building on previous work on Comanche [11], we proposed an approach to the problem of improving the calculation of out-of-core problems. Our extension of Comanche involves the use an out-of-core programming schema for optimizing data locality in nested loops to minimize I/O.

In our experiments, we have found that tiled stepped calculations without subarrays perform the best and use the least amount of memory space. We also have observed that the larger the data size is, the better the performance of tiled stepped calculations is. It obviously makes tiled stepped calculation an excellent out-of-core programming approach.

When applying tiled stepped concepts to multitasking (five stepped calculations are running concurrently for five iterations), the performance is dramatically better. Specifically, system performance degrades much more rapidly for VMM than for Comanche as more and more out-of-core applications are executed. Because tiled stepped calculations keep a small working data set in memory throughout the entire calculation, they use only 10.6% to 12.3% of the execution time of regular stepped calculations (refer to Figures 5.1, 5.2, and 5.3).

This illustrates our main issue with virtual memory management – its lack of information about the application’s actual memory needs and access patterns. As an application starts page faulting, the VMM system will take pages away from other applications even though those pages will only be used once and then discarded. Comanche uses the source code to determine the minimum amount of data it needs at any given moment in time and keeps this as the working set for the application.

![Figure 4.4 Performance of Tiled Stepped Calculations without Subarrays](image)

<table>
<thead>
<tr>
<th>Matrix Size</th>
<th>Elapsed</th>
<th>Page fault</th>
<th># of Seek</th>
<th># of Read</th>
<th># of Write</th>
<th>Memory usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>3072 x 3072</td>
<td>03:41.8</td>
<td>92,468</td>
<td>30,712</td>
<td>15,356</td>
<td>15,356</td>
<td>356,919</td>
</tr>
<tr>
<td>6144 x 6144</td>
<td>14:32.3</td>
<td>369,100</td>
<td>61,432</td>
<td>30,716</td>
<td>30,716</td>
<td>762,459</td>
</tr>
</tbody>
</table>

![Figure 5.1 Performance of multiple tasks regular stepped calculations for matrix size 4096 x 4096](image)

<table>
<thead>
<tr>
<th>Task Number</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page fault</td>
<td>320,481</td>
<td>313,200</td>
<td>309,686</td>
<td>308,346</td>
<td>315,001</td>
</tr>
</tbody>
</table>

![Figure 5.2 Performance of multiple tasks tiled stepped calculations with subarrays for matrix size 4096x4096](image)

<table>
<thead>
<tr>
<th>Block Size</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elapsed</td>
<td>09:56.0</td>
<td>09:56.0</td>
<td>09:56.0</td>
<td>09:56.0</td>
<td>09:56.0</td>
</tr>
<tr>
<td>Page fault</td>
<td>30,761</td>
<td>30,761</td>
<td>30,761</td>
<td>30,761</td>
<td>30,761</td>
</tr>
<tr>
<td># of Seek</td>
<td>40,944</td>
<td>40,944</td>
<td>40,944</td>
<td>40,944</td>
<td>40,944</td>
</tr>
<tr>
<td># of Read</td>
<td>20,472</td>
<td>20,472</td>
<td>20,472</td>
<td>20,472</td>
<td>20,472</td>
</tr>
<tr>
<td># of Write</td>
<td>20,472</td>
<td>20,472</td>
<td>20,472</td>
<td>20,472</td>
<td>20,472</td>
</tr>
<tr>
<td>Memory usage</td>
<td>1,435,295</td>
<td>1,435,295</td>
<td>1,435,295</td>
<td>1,435,295</td>
<td>1,435,295</td>
</tr>
</tbody>
</table>
Figure 5.3 Performance of multiple tasks tiled stepped calculations without subarrays for matrix size 4096x4096

References


