Application-Controlled Demand Paging for Out-of-Core Visualization

Michael Cox and David Ellsworth[1]
Report NAS-97-010, July 1997

NASA Ames Research Center
MS T27A-2
Moffett Field, CA 94035-1000

Abstract

In the area of scientific visualization, input data sets are often very large. In visualization of Computational Fluid Dynamics (CFD) in particular, input data sets today can surpass 100 Gbytes, and are expected to scale with the ability of supercomputers to generate them. Some visualization tools already partition large data sets into segments, and load appropriate segments as they are needed. However, this does not remove the problem for two reasons: 1) there are data sets for which even the individual segments are too large for the largest graphics workstations, 2) many practitioners do not have access to workstations with the memory capacity required to load even a segment, especially since the state-of-the-art visualization tools tend to be developed by researchers with much more powerful machines. When the size of the data that must be accessed is larger than the size of memory, some form of virtual memory is simply required. This may be by segmentation, paging, or by paged segments. In this paper we demonstrate that complete reliance on operating system virtual memory for out-of-core visualization leads to poor performance. We then describe a paged segment system that we have implemented, and explore the principles of memory management that can be employed by the application for out-of-core visualization. We show that application control over some of these can significantly improve performance. We show that sparse traversal can be exploited by loading only those data actually required. We show also that application control over data loading can be exploited by 1) loading data from alternative storage format (in particular 3-dimensional data stored in sub-cubes), 2) controlling the page size. Both of these techniques effectively reduce the total memory required by visualization at run-time. We also describe experiments we have done on *remote* out-of-core visualization (when pages are read by demand from remote disk) whose results are promising.

[1] Both authors are employees of MRJ Technology Solutions at NASA Ames Research Center.

Application-Controlled Demand Paging for Out-of-Core Visualization

Michael Cox
MRJ/NASA Ames Research Center
Microcomputer Research Labs, Intel Corporation
<mbc@nas.nasa.gov>

David Ellsworth
MRJ/NASA Ames Research Center
<ellswort@nas.nasa.gov>

Abstract

In the area of scientific visualization, input data sets are often very large. In visualization of Computational Fluid Dynamics (CFD) in particular, input data sets today can surpass 100 Gbytes, and are expected to scale with the ability of supercomputers to generate them. Some visualization tools already partition large data sets into segments, and load appropriate segments as they are needed. However, this does not remove the problem for two reasons: 1) there are data sets for which even the individual segments are too large for the largest graphics workstations, 2) many practitioners do not have access to workstations with the memory capacity required to load even a segment, especially since the state-of-the-art visualization tools tend to be developed by researchers with much more powerful machines. When the size of the data that must be accessed is larger than the size of memory, some form of virtual memory is simply required. This may be by segmentation, paging, or by paged segments. In this paper we demonstrate that complete reliance on operating system virtual memory for out-of-core visualization leads to egregious performance. We then describe a paged segment system that we have implemented, and explore the principles of memory management that can be employed by the application for out-of-core visualization. We show that application control over some of these can significantly improve performance. We show that sparse traversal can be exploited by loading only those data actually required. We show also that application control over data loading can be exploited by 1) loading data from alternative storage format (in particular 3-dimensional data stored in sub-cubes), 2) controlling the page size. Both of these techniques effectively reduce the total memory required by visualization at run-time. We also describe experiments we have done on *remote* out-of-core visualization (when pages are read by demand from remote disk) whose results are promising.

CR Categories and Subject Descriptors: D.4.2 [Operating Systems] Storage Management – storage hierarchies, segmentation, virtual memory; E.2 [Data] Data Storage Representations; J.2 [Computer Applications] Physical Sciences and Engineering – aerospace; I.3.2 [Computer Graphics] Graphic Systems – distributed/network graphics, remote systems, stand-alone systems, I.3.8 [Computer Graphics] Applications.

Additional Keywords: computational fluid dynamics, visualization, out-of-core visualization.

This is an extended version of a paper that appeared in *Proceedings of Visualization '97*, Phoenix AZ, October 1997.

1 Introduction

Visualization provides an interesting challenge for computer systems: data sets are generally quite large, taxing the capacities of main memory, local disk, and even remote disk. We call this the problem of *big data*. When data sets do not fit in main memory (*in core*), or when they do not fit even on local disk, the most common solution is to acquire more resources. This *write-a-check* algorithm has two drawbacks. First, if visualization algorithms and tools are worth developing, then they are worth deploying to more production-oriented scientists and engineers who may have on their desks machines with significantly less memory and disk. Some researchers have noted that their software tools were not used in practice for several years after development because the tools required more power and memory than were available on the average engineer's desk [15]. Second, there may not even be a machine that supports sufficiently large main memory or local disk for the data set one wishes to visualize. We find this in particular in the area of visualization of *Computational Fluid Dynamics (CFD).*

When a single data set is larger than the capacity of main memory, we must solve the problem of *out-of-core visualization.* When a single data set is larger than the capacity of local memory *and* disk, we must solve the problem of *remote out-of-core visualization*. We address primarily the first of these in this paper, although we also report what we believe are promising results from experiments in remote out-of-core visualization.

Out-of-core visualization requires *virtual memory* of some sort. We should be careful to distinguish between the *idea* of virtual memory, and the implementation(s) supported today by most operating systems (OSs). Virtual memory is simply the concept of mapping a larger *virtual* address space into a smaller *physical* space. Generally the larger virtual memory is partitioned into "pieces" each of which is moved into real memory when it is needed, at which time some "piece" that is (hopefully) no longer needed may be moved out. When the pieces are of fixed-length, the virtual memory is said to be in *pages* (or is said to be *paged*). When the pieces are of variable-length, virtual memory is said to be in *segments* (or is said to be *segmented*). When variable-length pieces are themselves partitioned into fixed-length pieces, virtual memory is said to be in *paged segments*. When pages or segments are loaded as they are needed, the system is said to be *demand driven* (e.g. *demand paged*). These are all well-studied schemes for virtual memory (cf. [16, 24]), and previous results and concepts from this area can be used productively for out-of-core visualization.

Perhaps the most well-known (often inadvertent) approach to out-of-core visualization is strict reliance on operating system virtual memory. To rely on the operating system for virtual memory support, the application allocates a buffer that is sufficiently large to hold the data set, and loads the data set into the buffer. If the data set is larger than physical memory, the operating system manages the discrepancy. The problem with this approach is that it generally results in poor performance due to *thrashing*. When a system *thrashes*, it spends more of its time replacing pages in physical memory with new pages from disk than it does accomplishing real work. We document this behavior in the current

paper, thrashing in CFD visualization has also been documented by Ueng [26]. Thrashing is more generally addressed in [2, 10, 14, 16, 18].

One approach to out-of-core visualization that has been more successfully employed than reliance on OS virtual memory is that of *application-controlled segmentation*. With this approach the application chooses some natural unit (segment) of data and specifically loads a segment when it is needed, possibly replacing some segment that is no longer needed. This is similar to the pre-virtual memory programming practice of *overlaying* code (data) segments with new code (data) segments as the former are no longer needed. Ueng et al. have successfully employed this approach with unstructured CFD data [26]. They *spatially* and hierarchically partition their data set in an octree, implicitly defining a segment to be a node of this tree. They load on demand each segment required by user-driven visualization, replacing the segments previously (but no longer) required. Kao[2] has successfully employed segmentation with primarily structured CFD data [19]. He *temporally* partitions the data set, implicitly defining a segment to be the data from one time-step of unsteady flow simulation. He sequentially loads each segment in order by time, calculating the visualization time-step by time-step, and replacing older segments as they are no longer needed.

While these purely segmented schemes have been successful, they are limited in several respects. First, the choice or computation of segment boundaries may be difficult, and in general may involve run-time parameters not available. In Ueng's approach octree decomposition is done off-line and the tree cannot be easily recomputed for machines with differing capacities. Second, if any segment (or group of segments) is significantly larger than the main memory of a given machine, the application reverts to strict reliance on operating system virtual memory. In Kao's approach a single temporal time-step may still exceed the capacity of main memory. For example, we are working with a structured grid data set at Ames for which a single time step comprises 550 Mbytes.[3]

However, we demonstrate in this paper that *application-controlled segmentation* can be productively augmented with *application-controlled demand paging*.

We first discuss the aspects of application-controlled memory management that may affect application performance. We then place our implementations and experiments in this context. Following this are out-of-core visualization experimental results, and some early *remote* out-of-core visualization experimental results. We then address related work, followed by conclusions and future work.

2 Application-controlled memory management

There are several principles of memory management that an application can exploit to improve performance. We discuss these briefly before describing the implementations and experiments we have performed to explore the issues in the context of CFD visualization. Following this we discuss results and the memory management issues in Section 4.

[2] Who previously went by the name Lane.

[3] The simulation is currently steady, with work ongoing to generate an unsteady data set of the same aircraft.

2.1 Sparse traversal

We should expect many visualization algorithms to traverse only a subset of the entire data set. If we assume for example that traversal of each cell of 3-dimensional data results in the generation of geometry, traversal of the entire data set would generate geometry to fill 3-space, making the image visually difficult to comprehend! There are of course algorithms that today must traverse the full data set, but we argue in [9] that finding the algorithm with the most *parsimonious traversal* is an important step in out-of-core visualization.

The most common approach today is to pre-load the entire data set before traversing it for visualization. If traversal really is sparse, more data are touched than need be. In particular if the data do not fit in physical memory, some of the data must be unnecessarily paged by the operating system. As an alternative, we may load *only* those data that *are* required *as* required. If not all of the data are loaded, we say that the application takes advantage of *sparse traversal*. This demand-driven strategy may be based on segments (as in [26]) or on pages. In this paper, we report our extension of Kao's temporal segmentation with demand paging in order to support sparse traversal.

2.2 Replacement policy

When more data are required than fit in physical memory, new data must supplant old data. In general, as each virtual page of new data must be brought into physical memory, some virtual page that is already *resident* must be chosen as a *victim* for *replacement*. The policy by which a victim is chosen is called the *replacement policy*. The ubiquitous replacement policy in operating systems today is *Least Recently Used (LRU).* That is, the page of the application that has been accessed the least recently is selected as a victim.[4] We have explored application-controlled replacement, but have not found strategies competitive with OS-controlled replacement. In this paper we describe implementations that leave page replacement to the operating system.

2.3 Load/store management

With reliance on the virtual memory of today's operating systems, the movement of data between memory and disk is under OS control. This leads to lost opportunities.

2.3.1 Page size

Commercial operating systems today support only fixed page sizes (typically 4 or 16 Kbytes). Application-controlled variable page sizes have been explored by researchers in operating systems but the results have not propagated to widely used systems. The problem with fixed page size is that the choice made by the operating system may not be the right one for all applications. In particular, the *granularity* of the page may be too coarse. As example, consider a hypothetical visualization that requires a small cluster of data from the center of a large cube of data. Suppose further that the cube is partitioned into 8 large pages so that each page holds some of the clustered data, forcing the application to require the entire data set. Repartitioning the cube into 64 smaller pages in general will force the cluster onto fewer pages, allowing the visualization to require less memory. In this paper we report experimental results concerning page size for CFD visualization of structured grids.

[4] In practice, the replacement policy is typically more complicated, also involving physically mapped pages from other applications, disk cache, etc.

2.3.2 Translation

Multi-dimensional scientific data are often represented in program code as multi-dimensional arrays. These arrays have traditionally been stored in memory in row- or column-order. That is, they are stored first linearly along one dimension, and then along a second, and then along a third. The program typically accesses the multi-dimensional array by indexing, e.g. *array[i][j][k]*. The compiler translates this reference by multiplying by the appropriate strides in the array, and generating a virtual address that is an offset from the beginning of the array. The operating system then translates the virtual address to a physical address. However, multi-dimensional scientific data tend to be accessed coherently in three dimensions, in particular as the result of 3-dimensional traversal. In volume rendering for example, it is well known that storage in "cubes" results in more efficient access than storage in planes (cf. [23]). We distinguish the traditional *flat storage* (row- or column-order) from this alternative *cubed storage*. To introduce support for cubed storage into applications written for flat storage would require source code modification to use something other than array references. Instead, we would like to provide (as transparently as possible) translation from the application's *flat* array references to alternative data organizations in physical memory (e.g. *cubed*).

2.3.3 Loading

Support for alternative data storage organizations may require additional processing when data are loaded from disk. For example, when a larger cube of data is partitioned into smaller sub-cubes, there is generally *internal fragmentation* within the sub-cubes. Internal fragmentation is the loss of memory within a page because of inefficiency in packing. In this case, internal fragmentation arises whenever the dimensions of the sub-cubes do not evenly divide the dimensions of the larger cube. When this occurs, sub-cube pages must be padded to align with the larger cube boundaries so the addresses of sub-cubes can be calculated in closed form. We call the result of such padding *file bloat*. In CFD data sets in particular, we have found that bloat can result in files 200% larger than their flat counterparts.[5] When data sets can be several hundred Gbytes, such expansion of file size is simply unacceptable and it is clear that cubed files must be packed or compressed on disk. In general, such packing requires resort to variable-length pages. While it is fairly easy to support storage and look-up of variable-length pages on disk, it is much more challenging to support reference by reference access to variable-length pages in memory.[6] The obvious solution is to pack cubed files for more efficient storage on disk, and unpack them when pages are loaded into memory. The virtual memory primitives of today's operating systems do not support such translation. In this paper we report performance improvements that can be achieved when application control over data loading is used to support access to packed cubed files. We support such access by translating the original array references in the application to variable-length pages on disk that we then unpack into fixed-length pages in memory.

[5] This is true in particular of structured grids in CFD because each data set generally comprises many smaller grids, called *zones*.

[6] In particular without hardware and operating system support.

In many visualization applications there are *derived data* (or *derived fields*) that are not stored with the data set – rather they are derived at run-time. In general the entire derived field may be *eagerly* calculated so that data are available when needed, or the derived field may be *lazily* calculated only as pages are required during traversal. If the data set is enormous, eager evaluation is more difficult than out-of-core visualization of the underlying data! Alternatively, if the application has sufficient control over data loading, derived data may be lazily calculated only when each page is loaded. As with the underlying data, if the traversal is sparse fewer pages need be calculated and managed. Although we note demand paging of derived data as a promising direction, we do *not* in the current paper report implementation or experimental results.

2.3.4 Storing

When the application loads a page of data from disk into memory, the OS marks the page so that it will be later saved (i.e. the OS marks the page as *dirty*). The OS does this because from its point of view, the page has been written by the application.[7] When the underlying physical memory is subsequently required for another virtual page (i.e. the virtual page must be *replaced*), the OS saves the data from the dirty page to disk for subsequent re-use. This results in inefficiency for two reasons: 1) if the data were originally read from disk, they need not be stored since they can be re-read from the original file, 2) the data may not be required again anyway, since a visualization application's traversal may not revisit the same cell or cells. Ideally, the application would control which data were stored when virtual pages were replaced, and which data were simply discarded. Current operating systems do not support this.

In the current paper, we describe an implementation that unfortunately cannot take advantage of these opportunities (because storage and page replacement are inextricably linked). However, we believe performance improvements are available with more application control over both policies. This is further discussed in section 8.

3 Experimental methodology

Before discussing experimental methodology, we first review CFD visualization and the original implementation of the software package that we have used as test-bed – the Unsteady Flow Analysis Toolkit (UFAT) [19]. Following this we discuss an implementation of UFAT modified to use the Unix system call *mmap()* in order to demonstrate the performance benefits of sparse traversal. Then we describe a user-level demand-paging implementation of UFAT we use to explore application control that is not supported by *mmap()*. We finish this section with sundry details of experimental methodology.

3.1 Visualization of Computational Fluid Dynamics

Computational Fluid Dynamics (CFD) visualization systems must process input data of several types, with some complexities. The data may or may not be on a regular lattice (*structured* if they are, *unstructured* if they are not). Furthermore, the coordinates of the nodes of the lattice generally do not correspond to actual coordinates in space. Coordinates in the lattice are generally referred to as *computational space*, and the real locations to which

[7] Even if the page is memory-mapped, the OS generally takes this conservative approach so that it need not guarantee consistency on the underlying file.

they correspond are generally referred to as *physical space* (cf. [1, 6]). To implement these two spaces, the values at nodes in the lattice are generally provided in one input file (*solutions*), and the node-by-node mappings to physical space are generally provided in another input file (*grid*). Each grid itself may comprise multiple sub-grids, and each of these is generally referred to as a *zone*. Furthermore, while there may be only one solution if the flow is at equilibrium (*steady*), multiple solutions may be input if the flow is time-varying (*unsteady*), and multiple grids may be input if the mapping to physical space is itself time-varying (that is, if the grid itself changes over time).

The algorithms used to visualize CFD data include *streamlines*, *streaklines*, *particle traces*, *vortex-core finding*, as well as the *cutting planes*, *isosurfaces*, and *local isosurfaces* employed in other application domains. Most of these are local algorithms that only need to traverse a subset of the data in order to calculate the synthetic geometry for a given visualization. Most CFD visualization systems have supported visualization of steady flows (single grid, single solution input) (e.g. [4, 5, 28]). These have typically avoided the problem of big data by requiring that both the grid and solution fit entirely in main memory before the visualization begins.

At least one system supports unsteady flow visualization (multiple grids, multiple solutions) – the Unsteady Flow Analysis Toolkit (UFAT) [19]. Aside from the algorithmic challenges that must be tackled to visualize flow through multiple time steps, unsteady flows challenge the computer system with significantly big data. Typically the "solver" outputs only 1 in 10, or 1 in 100 of the time steps due to limited system, disk, and visualization system capacity. But even then, the output may have hundreds of time steps, each of which may today surpass 500 Mbytes.

3.2 Unsteady Flow Analysis Toolkit (UFAT)

UFAT has implicitly employed segmentation to handle such potentially large time-varying data sets. In UFAT, each grid for each time step is (implicitly) defined as a segment, as is each solution for each time step. UFAT explicitly interpolates between a fixed number of time steps at once, and so when the time step is advanced, the oldest solution segment is overlaid with a new solution segment, similarly for grid segments. However, UFAT loads an entire segment (or pair of segments) when it advances time steps, and so before it is through loads the entire data set. In addition, if the grid plus solution data required for any new time step are larger than physical memory, UFAT relies on operating system virtual memory, and its performance drops precipitously. Finally, UFAT reads primarily PLOT3D data files [28], which employ *flat storage* of data. In the remainder of this paper, we call this implementation of UFAT the *original UFAT*.

3.3 Mapped UFAT

As discussed, the original UFAT employs application-controlled segmentation. When UFAT traces, say, particles through an unsteady flow, it calculates the hypothetical paths of massless particles through the flow over time. At any point in the calculation, it loads into memory the data for times *t* and *t+1*. When UFAT advances the time step to *t+2*, it reuses the buffer it used for time *t*. Now, if UFAT actually required all of the data during each time step and there were insufficient physical memory, it would be difficult to prevent thrashing. On the other hand, if the traversal through the data were sparse, it would be advantageous to load only those pages actually touched. In order to demonstrate specifically the advantage of sparse traversal, we have modified the original UFAT to memory map input files (with the Unix system call *mmap()*) rather than read them explicitly into memory. The result is that a page from disk is only read into physical memory when accessed. If only a fraction *f* of the data is required during traversal, then only the fraction *f* is read from disk into memory. At the end of processing of each segment, mapped UFAT *unmap()*'s the segment, effectively freeing the underlying mapped pages for subsequent reuse. However, *mmap()* does not offer the application control over page size, nor does *mmap()* provide the semantics that would be required to support translation of array references to packed cubed data files. We call the *mmap()'d* implementation *mapped UFAT*.

3.4 Paged UFAT

In order to explore the advantages of additional application control over memory management (in particular translation and page loading), we have implemented in user-space a demand paging system that takes control over some of the paging functions of grid and solution input to UFAT. The details of this implementation are discussed below. We call this implementation *paged UFAT*.

Paged UFAT implements demand paging of segments in a way similar to mapped UFAT. When a new segment is "loaded", we simply "map" the data contents (without loading data). Then, as data in the underlying segment are demanded, we allocate physical pages and read the underlying data from disk. Paged UFAT differs from mapped UFAT in several respects. First, page size is a configurable parameter, allowing us to explore its effect on performance. For any given size however, physical page size must be the same in memory as on disk. Second, paged UFAT explicitly allocates a pool of free pages for grid and solution data, in contrast to mapped UFAT which treats all of physical memory as its pool. This pool is partitioned into pages of the desired size. If the pool is empty, we allocate additional memory and partition this into new free pages. Third, as UFAT references data that are not resident, we request a free page from the data pool, and explicitly load and unpack the data from disk into that page.

On the other hand, paged UFAT is similar to mapped UFAT in that page replacement (when data requirements exceed physical memory) is handled by the operating system, and in that all allocated pages are returned to the "pool" after a segment is processed.

3.4.1 Translation

For any general demand paging system, it is really a requirement that the application be allowed to reference underlying data via its native virtual addresses. In our case, the underlying UFAT code references data in *computational coordinates* (i.e. as *array[i,j,k,field]*). While this works when the data are laid out in memory in row- or column-order, it does not work when the underlying storage of the data are not flat (e.g. when the data are stored in cubed format).

In order to support alternative underlying data storage (and also to support compression on disk of the underlying data), we translate the virtual computational coordinates into the underlying "file coordinates". There are several steps in this translation. First, when UFAT references the underlying data (as *array[i,j,k,field]*), we trap the call (by trapping the array reference at the FORTRAN call site to a function call of the same name). If the underlying page is resident, we simply return the data. Otherwise, we translate the array reference into a *virtual block address* in the underlying file. This translation differs depending on whether the underlying storage is flat or cubed. We translate to *virtual* block address

because (as previously discussed) there may not be a one-to-one mapping between storage in memory and storage on disk. From virtual block address, we translate to physical offset within the file, then allocate a page from the free pool and read the data into memory. Once the page is resident, execution proceeds as it would otherwise have, and we return the data originally requested by the multi-dimensional array reference.

It is probably clear from this discussion that in the user-space implementation of paged UFAT translation is very expensive. We have corroborated this expectation with profiling and have found that our address translation consumes more CPU cycles than any other UFAT routine. For example, for the shuttle data set (described below) address translation initially accounted for 80% of CPU utilization. We have made a first pass at alleviating this high cost by taking advantage of the fact that for most array references, neighbors are also soon referenced. We have added new translation routines that return several values instead of one value, which amortizes the translation cost over several cells. This approach has reduced the percentage of execution time taken by address translation, and was enabled during the experiments reported below. But still, even with this technique, address translation on the shuttle accounted for 50% of CPU utilization in the runs discussed below. We consider the positive results we report even stronger in light of this cost.

3.4.2 Loading

Application control over loading is important in a number of contexts: 1) when storage in memory does not correspond to storage on disk, 2) when pages may be loaded from non-file sources, 3) when the application chooses *lazy* rather than *eager* evaluation of derived data (e.g. when a derived field is defined over an entire data set, but visualization is only desired of some limited traversal through the data set).

In the current paged UFAT, we take advantage only of the first of these opportunities. In particular, when the underlying file storage is cubed, regular addressing results in "holes" in the underlying stored file. These holes can result in 200% bloat if the underlying data are not "packed". We support packed files by storing with each cubed file a *block translation table* which provides the physical offset of the block within the file and the number of bytes that the block actually comprises. The *virtual block address*, then, is used to index this table to find the actual block. When the block is read into memory, a full memory page is allocated regardless of the actual underlying block length, and any unused memory in the page is left uninitialized and undefined.

3.5 Experimental methodology

We have employed the data sets shown in Table 1. The experiments we performed on these data sets were chosen to emulate (or replicate) studies for which the data were originally used, as described below.

Tapered cylinder. It is well known that the behavior of vortices on the downstream side of flow past a cylinder is a function of cylinder diameter. The tapered cylinder was designed to explore the vortex behavior on the downstream side of a cylinder of continuously varying diameter [17]. We have introduced per-timestep streamlines on the back side of the cylinder to replicate (and exaggerate) the original experiments. All frames (concatenated into one) from the particle trace are shown in Color Plate 1.

Name	Tapered cylinder	Shuttle	F18	High-wing
Type	Unsteady Single-zone	Steady Multi-zone	Unsteady Multi-zone	Steady Multi-zone
Time steps	100	1	220	1
Grid (Mbytes)	1.5	14.4	26.9	246.4
Solution (Mbytes)	2.5	18.0	33.7	308.0
Total (Mbytes)	251.5	32.3	7432.0	554.4

Table 1. Data sets.

Name	Tapered cylinder	Shuttle	F18	High-wing
Machine	Indigo2	Indigo2	Onyx2	Indigo2
OS	IRIX 6.2	IRIX 6.2	IRIX 6.4	IRIX 6.2
Disk (Gbytes)	4	4	14	4
Memory (Mbytes)	128	128	1024	128

Table 2. Test environment. Indigo2 and Onyx2 are 195 MHz R10000's. All disks are standard SCSI.

Shuttle. One study conducted on the shuttle was done to determine the behavior of debris that might collide with the it [6] We have emulated this study by introducing a rake of particles at the front of the shuttle's fuel tank, and traced streamlines. The results are shown in Color Plate 2.

F18. Some of the studies conducted on the F18 focused on vortex behavior beginning above the wing and proceeding to the rear of the plane [13]. We have emulated some of the traces of this study by introducing particles in and around the vortex core above the wing, and calculating streaklines. The concatenated results from the 220-frame animation are shown in Color Plate 3.

High-wing. The high-wing data set is under commercial non-disclosure, and we are not able to publish a picture at the time of this writing. However, one rake of streamlines was placed before the wing and two were placed at engine exhausts. Among the three rakes a trailing vortex was captured. This particular experiment was illuminating for the principle investigators on the project at NASA Ames Research Center.

All of our code has been based on Version 3.0 of UFAT. We compiled all code with the SGI C compiler with flags "–n32" and "–O2". We performed experiments using the machines shown in Table 2. Original UFAT and mapped UFAT were provided input from PLOT3D files, paged UFAT was provided input from our own file format that supports both flat and cubed storage. PLOT3D files were automatically translated to our file format, and after visualization with paged UFAT, graphical output files were compared for equivalence.

In order to study the effects of limited available memory, we used the system call *mpin()* effectively to remove memory from each of the machines above. In a separate process that began before and ran during each limited memory experiment, we allocated and pinned sufficient memory to reduce the total memory available on the workstation to the desired target. This target is the "memory capacity" reported in the results in section 4. To compensate for differences in kernel size between the actual machine and target machines, we scaled slightly the amount of memory pinned.

However, it is important to note that all runs were subject to the same memory environment.

Between all runs, we cleared the file cache by allocating the bulk of memory available in user-space (which has the effect in IRIX of reducing the pages available to the file cache) and by subsequently randomly reading a file the size of the target machine's memory.

4 Out-of-core visualization

Summaries of results are shown in Tables 3 through 6. The data set is identified in the caption, the run in the leftmost column, and the memory capacity (in Mbytes) is identified in the topmost row by *M=*. We have explored the performance of original UFAT (*Original*), mapped UFAT (*Mapped*), and paged UFAT with a range of block sizes and with cubed and flat storage. These are labeled as *N-cubed* and *N-flat* where *N* corresponds to the following block and page sizes:

N	Cube dimensions (cells)	Page size (bytes)
4	4 x 4 x 4	256
8	8 x 8 x 8	2K
16	16 x 16 x 16	16K
32	32 x 32 x 32	128K

4.1 Notes on the experiments

The tapered cylinder was only run in an environment of *unlimited memory* because it is a small data set. The shuttle was run in environments between unlimited memory (*M=128*) and *limited memory* (*M=32*). The F18 was run between unlimited memory (*M=1024*) and *very* limited memory (*M=32*). The high-wing is a 554 Mbyte data set that researchers at NASA Ames currently wish to explore using their desktop machines; hence, we have explored its behavior only with progressively more limited memory (between *M=128* and *M=32*). *Results for original UFAT are not available for the high-wing, because that application died due to insufficient swap space* (the machine's configuration was standard – twice the swap space of main memory).

4.2 Overall results

Overall, performance of paged UFAT with 8 x 8 x 8 cubes (2K pages) surpassed that of either original UFAT or mapped UFAT. Over all block sizes and over both storage formats (cubed and flat), 8 x 8 x 8 cubes generally provided the best performance amonged paged UFAT runs as well. The tapered cylinder is the most notable exception, where flow is anomalously axis-aligned, and where flat storage results in better performance.

In addition, paged UFAT degraded gracefully with decreasing available memory, as can be seen in Figures 1 through 3. At the same time, mapped UFAT degraded faster than paged UFAT, and original UFAT *clearly* did not degrade gracefully. *These results strongly suggest that out-of-core visualization cannot be achieved simply by loading all data and relying on operating system virtual memory.*

Run	M=1024	M=128	M=64	M=32
Original	—	115.9	—	—
Mapped	—	116.6	—	—
4-cubed	—	83.6	—	—
4-flat	—	86.7	—	—
8-cubed	—	94.9	—	—
8-flat	—	67.3	—	—
16-cubed	—	107.4	—	—
16-flat	—	62.3	—	—
32-cubed	—	85.4	—	—
32-flat	—	71.76	—	—

Table 3. *Tapered cylinder* experimental results (seconds).

Run	M=1024	M=128	M=64	M=32
Original	—	8.5	14.9	25.7
Mapped	—	9.8	9.1	13.8
4-cubed	—	11.0	11.6	15.2
4-flat	—	10.2	11.6	22.4
8-cubed	—	8.6	9.3	11.0
8-flat	—	11.4	11.8	24.9
16-cubed	—	8.8	8.2	15.3
16-flat	—	8.4	10.6	22.0
32-cubed	—	8.2	10.0	21.3
32-flat	—	8.2	8.7	21.6

Table 4. *Shuttle* experimental results (seconds).

Run	M=1024	M=128	M=64	M=32
Original	1051.6	1080.0	3369.7	5704.0
Mapped	588.8	592.3	620.8	843.5
4-cubed	392.3	414.2	478.5	598.8
4-flat	642.6	673.8	860.2	1167.4
8-cubed	326.5	331.5	372.8	462.2
8-flat	615.0	640.0	764.7	1094.3
16-cubed	387.6	391.8	434.3	611.8
16-flat	710.0	724.6	925.3	2138.6
32-cubed	501.1	502.6	826.0	1982.4
32-flat	611.9	602.2	872.4	2055.9

Table 5. *F18* experimental results (seconds).

Run	M=1024	M=128	M=64	M=32
Original	—	**N/A**	**N/A**	**N/A**
Mapped	—	111.1	247.7	671.1
4-cubed	—	118.6	243.2	331.3
4-flat	—	168.3	273.3	527.5
8-cubed	—	81.7	129.3	248.6
8-flat	—	131.8	247.5	786.2
16-cubed	—	117.8	163.5	354.9
16-flat	—	145.3	270.0	543.5
32-cubed	—	148.8	339.9	899.8
32-flat	—	151.7	370.1	817.3

Table 6. *High-wing* experimental results (seconds).

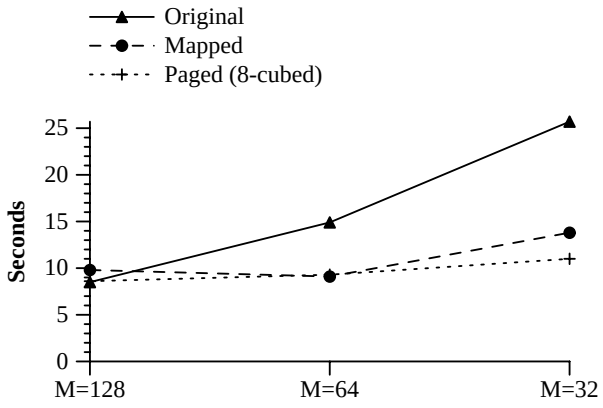

Figure 1. Comparative performance, *Shuttle*.

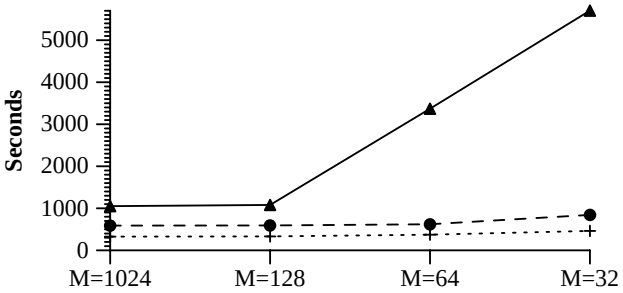

Figure 2. Comparative performance, *F18*.

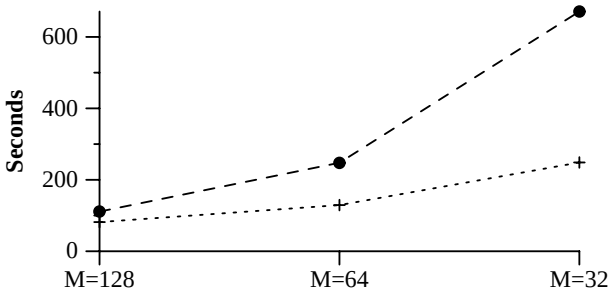

Figure 3. Comparative performance, *High-wing*.

4.3 Sparse traversal

In order to provide a common basis of comparison, we compare the original, mapped, and paged UFATs when all utilize the same page size and flat storage. The original and mapped versions naturally employ 16K physical pages, while paged UFAT does so when *N=16*. Note from Tables 3 through 6 that the demand paged systems that can take advantage of sparse traversal virtually always perform better (and when not, they are roughly on par). This is shown more graphically in Figures 1 and 2 for the shuttle and F18. Note from these figures that it is even more important to take advantage of sparse traversal as memory becomes more limited.

Finally, note that when paged UFAT works with the same 16K pages (16-flat) that mapped UFAT must work with, paged UFAT is generally slower. This is because of the overhead that paged UFAT incurs in managing memory in user-space, without hardware and operating system support. The fact that cubed storage and smaller pages can make paged UFAT *faster* is all the more compelling evidence that these are important performance issues for out-of-core visualization.

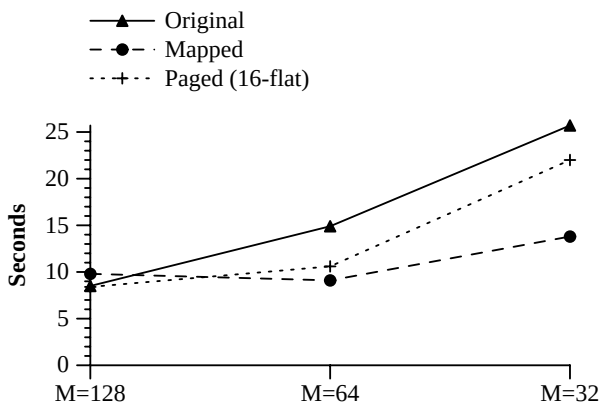

Figure 4. Sparse traversal in *Shuttle*.

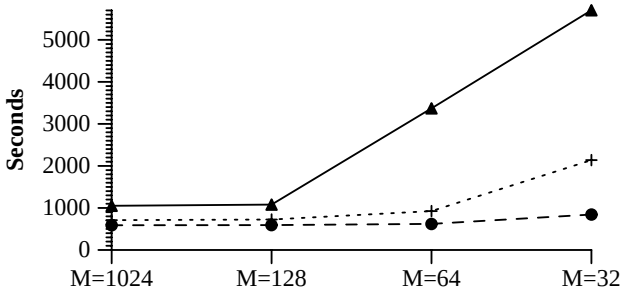

Figure 5. Sparse traversal in *F18*.

The percentage of blocks touched during visualization bears out the hypothesis that streakline and streamline traversal in CFD visualization is sparse. Below are shown the percentages of blocks touched by paged UFAT for 8 x 8 x 8 cubes:

	Percentage of pages touched		
	Grid	Solution	Overall
Tapered cylinder	42.6	17.2	23.1
Shuttle	27.9	6.3	15.9
F18	15.5	2.3	6.1
High-wing	18.9	2.3	9.7

Sparse traversal can also be seen in the *working sets* of Figures 6 and 7. The working set is defined as the set of blocks required during some period of time (in our case a single time step). In these are graphed the fractions of grid and solution pages required during traversal. The darker lines show cubed working sets, the ligher lines show flat working sets. These pictures confirm that generally only a fraction of the pages are required. However, several additional observations deserve note. First, grid working sets are "peaky", surpassing 50% of the total pages at times. This is because the current algorithms in UFAT search the grid when streaklines or streamlines cross zone boundaries. Second, there are clearly patterns of access where it should be possible to exploit better page replacement strategies than LRU. We do not do so in the current paper but mention it in passing. Third, these graphs make it clear that the working sets of cubed files are smaller than they are for flat files, and that at least for solution data smaller page size leads to reduced working set size. These are the topics of the next two sections.

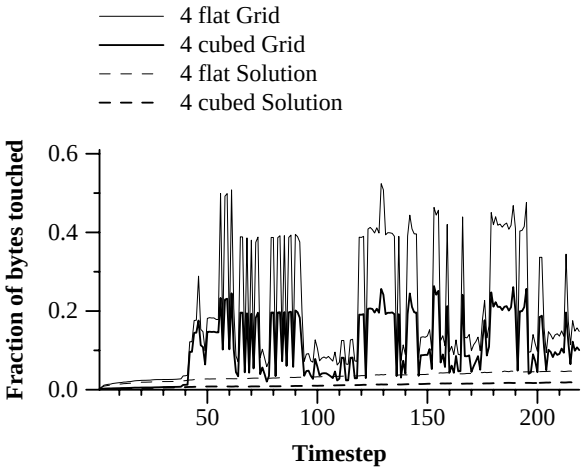

Figure 1. Working set for *N=4, F18*.

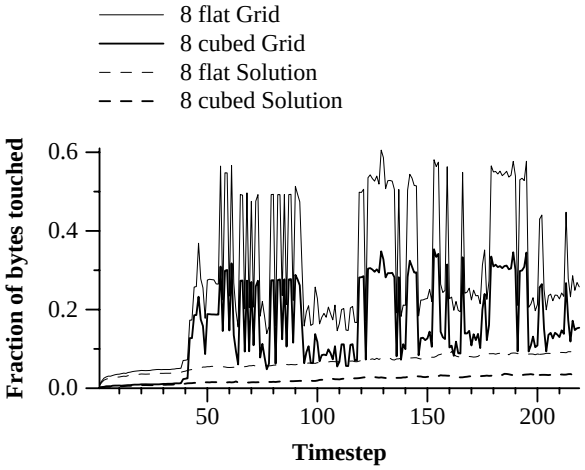

Figure 2. Working set for *N=8, F18*.

4.4 Load/store management

As discussed in section 3.4.2, paged UFAT takes control over two aspects of page load/store management, in particular translation and page loading. By doing so, it can support alternative page sizes, and alternative page storage formats (without wasting disk storage with bloated files). Using the standard system services available today, in particular *mmap()*, it is not possible to support these features. In this section we examine the performance improvements that they provide.

4.4.1 Cubed vs. flat storage

It is clear from Figures 6 and 7 and from inspection of Tables 3 through 6 (especially for the F18 and high-wing) that for fixed page size, cubed storage is generally significantly better than flat storage. This is because for most 3-dimensional traversals, cubes provide better locality of reference than do planes. As a result, fewer pages are required at run-time. This trend is most noticeable for all runs as memory pages become scarcer. Cubed storage allows the application to take better advantage of the pages that are available.

Finally, we make two observations. First, without support for translation to cubed format from linear array references, the application cannot take advantage of such storage without modification. Second, without support for application processing during page loading (which *mmap()* does not provide) *packed* cubed files cannot be supported.

4.4.2 Page size

There are two competing forces affecting the dependency of performance on page size. Paged UFAT requires its own internal page tables in order to manage its own demand paging. As blocks become smaller the page tables grow, themselves consuming memory. On the other hand, smaller blocks allow finer granularity and in general result in a smaller working set. In our experiments, we have found the cross-over of these two curves with 2 Kbyte pages (8 x 8 x 8). This trend is shown graphically for the F18 in Figure 8.

Finally, the standard page size is 16 Kbytes on the machines and operating systems we used as platform. We note that without application control over loading pages, our smaller pages would not have been employable.

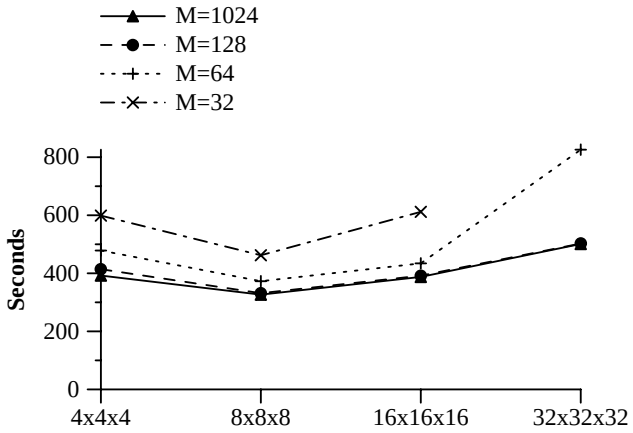

Figure 3. Page size sensitivity for cubed storage, *F18*. (32x32x32 at *M=32* deliberately omitted because of its effect on graph scale).

5 Remote out-of-core visualization

The second problem we address in this paper is remote out-of-core visualization – the local visualization of data sets that are stored remotely because they do not fit on local disk. In this model a file server provides pages to smaller local workstations on demand. To explore the viability of this architecture, we have run paged UFAT and mapped UFAT on the same mid-range workstation as in the Table 2, with the high-wing stored on remote server accessible via the Network File System (NFS) over 10 Mbit Ethernet. Ideally, this architecture would be supported at least by 100 Mbit Ethernet. However, even over the slower network link the results are encouraging. The results for the high-wing are shown in Table 7.

	M=128		M=64		M=32	
	Loc.	Rem.	Loc.	Rem.	Loc.	Rem.
Paged	81.7	183.3	129.3	206.9	248.6	259.0
Mapped	111.1	261.8	247.7	347.0	671.1	1104.1

Table 7. Remote demand paging for paged UFAT (cubed 8 x 8 x 8) and mapped UFAT. Local (*Loc.*) and Remote (*Rem.*) times are shown (seconds).

As can be seen, the degradation for paged UFAT is at worst somewhat greater than a factor of 2, and for very limited memory (*M=32*) remote and local are essentially at parity.

6 Related work

Researchers in operating systems have recently explored extensions to standard systems to support more application control over virtual memory. The case for these extensions has been made repeatedly (cf. [2, 10, 14, 18, 21, 27]). Some research prototypes have added more application control [11, 14, 20, 21, 27] but these features have unfortunately not found their way into commercial operating systems. Appel and Li have demonstrated by operating system modification that application control over write-back policies can improve performance by discarding dirty data that really are garbage or that can be rederived [3]. Just such control would be desirable for visualization where data can be re-read from disk. Cao et al. have explored application control over file caching [7, 8]. Their focus has primarily been on efficient implementation and on global performance.

In the visualization domain, Song has demonstrated that the problem of big data can be mitigated in a data flow system by reducing the granularity of data flow nodes [25]. For visualization of earth sciences data, the Common Data Format (CDF) library [22] implements a simple form of demand-paged segments. In our terminology, CDF maps a segment to each file, and independently demand pages each of these segments. Since a cache is associated with each file, the memory in use grows with the total number of open files. Application control over this growth is difficult unless the application keeps track of its own access patterns on the underlying data. We are unaware of studies on CDF that explore alternative page sizes, replacement policies, and data storage and organization, and so cannot address the trade-offs in demand-paged segments for earth sciences data.

In a different visualization domain, Funkhouser explicitly used segmentation to explore architectural databases at interactive rates [12]. He partitioned objects hierarchically in an approach similar to the one taken by Ueng for unstructured CFD data [26]. He was able to visualize at interactive rates a database roughly 10x the size of main memory. While these results and the techniques they suggest are of interest, the differences with respect to scientific visualization should be explicitly noted:

- With synthetic imagery, data traversal is driven by direction of travel of a viewer; in scientific visualization data traversal is driven by the visualization algorithm (and is generally unrelated to the viewer) and geometry is not generated until after traversal.
- With synthetic imagery, data that will not be needed can be explicitly culled by fairly well-known algorithms; in scientific visualization, it is not yet clear which data can be culled and which data cannot be culled (and in any event is visualization algorithm-specific).
- The sizes of the biggest synthetic data are significantly smaller than those encountered in scientific visualization.

7 Conclusions

When a single data set that we wish to visualize is larger than the capacity of main memory, we must solve the problem of *out-of-core visualization*. When a single data set is larger than the capacity of local memory *and* disk, we must solve the problem of *remote out-of-core visualization*. We have addressed primarily the first of these in this paper, although we have reported what we believe are promising results from experiments in remote out-of-core visualization.

To tackle out-of-core visualization, we have built upon a previous technique to limit the size of data that must be in core at any time, in particular *segmentation*. Previous authors have used application-controlled segmentation. In particular, they have partitioned their data sets along natural boundaries, defining each subset as a segment, and loaded segments only when they were needed. We have added application-controlled demand paging to a previous segment-based system, and in doing so have demonstrated significantly better performance than previously achieved by simple reliance on operating system virtual memory. Furthermore, we have demonstrated better performance not only when data size exceeded physical memory (*limited memory*) but also when physical memory was sufficient to hold the data (*unlimited memory*).

The principles we have exploited can be summarized as follows:

Sparse traversal. When only a subset of the data are required for a given visualization, demand loading only those pages necessary leads in general to better performance. When memory is limited demand paging is even more important to sustain acceptable performance. We have found that even with unlimited memory, demand paging leads to better performance than loading the entire data set.

Page size. The finer the grain of page size, the fewer pages required for given traversal. We have found the best overall performance with page sizes smaller than those supported by the standard operating system(s).

Cubed storage. When data are stored in "cubes" rather than in flat planes, there is generally better locality of reference. Improvement in locality reduces the number of pages a visualization application requires at run-time. We have found that cubed storage results in significantly better performance than flat storage. However, cubed storage generally leads to larger files (by as much as a factor of 2). To solve this, we have translated at run-time from a packed file representation on disk to an unpacked representation in memory. This has allowed us to support cubed storage with minimal increase in disk storage requirements.

We note that exploitation of the second two of these requires memory management support not present in today's operating systems. This suggests that for the near term, out-of-core visualization will require support by user-level memory management.

We have also explored *remote* out-of-core visualization (where demand paging is from a remote data server). Our results are promising, showing only roughly a factor of 2 slow-down over our best local out-of-core visualization.

8 Future Work

We intend to explore approaches that draw more support for application-controlled memory management from the operating system. We also believe there is opportunity to take advantage of additional techniques to improve out-of-core visualization performance, in particular prefetching and data set indexing. There may be other visualization applications that can exploit demand-paged segmentation, and we welcome collaboration in exploring other domains. Finally, we believe that remote out-of-core visualization is a very promising approach to provide visualization tools to a broader user community.

9 Acknowledgements

The authors are grateful to David Kenwright whose help with CFD visualization and its tools has been indispensable. The authors would like also to thank Scott Murman and Ken Gee for allowing us to use and helping us to acquire the F18 data, Karlin Roth for use of the high-wing data set, David Kenwright and David Kao for help with UFAT, Sandy Johan for help acquiring the data sets, and Bryan Green for help with the machines we required to process the F18 data set. We thank David Kao for his *mmap()* code that we dusted off and refurbished for the current paper.

10 References

1. J. Anderson, *Computational Fluid Dynamics: The Basics with Applications*, McGraw-Hill, New York NY, 1995.
2. T. Anderson, "The Case for Application-Specific Operating Systems," *Proceedings of the Third Workshop on Workstation Operating Systems*, April 1992, pp. 92-94.
3. A. Appel and K. Li, "Virtual Memory Primitives for User Programs," *Proceedings of the 4th Symposium on Architectural Support for Programming Languages and Operating Systems*, Santa Clara CA, April 1991.
4. G. Bancroft et al., "FAST: A Multi-Processed Environment for Visualization of Computational Fluid Dynamics," *Proceedings of Visualization '90*, San Francisco CA, October 1990, pp. 14-27.
5. S. Bryson and C. Levit, "The Virtual Wind Tunnel," *IEEE Computer Graphics & Applications*, Vol. 12, No. 4, July 1992, pp. 25-34.
6. P. Buning et al., "Flowfield Simulation of the Space Shuttle Vehicle in Ascent," *Fourth International Conference on Supercomputing*, Vol. II, Supercomputer Applications, Kartashev & Kartashev, eds, 1989, pp. 20-28.
7. P. Cao, E. W. Felten, and K. Li, "Application-Controlled File Caching Policies," *Proceedings of the 1994 Summer USENIX*, June 1994.
8. P. Cao, E. W. Felten, and K. Li, "Implementation and Performance of Application-Controlled File Caching," Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation (OSDI), November 1994, pp. 165-177.
9. M. Cox and D. Ellsworth, "Managing Big Data for Scientific Visualization," *ACM SIGGRAPH '97 Course #4, Exploring Gigabyte Datasets in Real-Time: Algorithms, Data Management, and Time-Critical Design*, ACM SIGGRAPH '97, August 1997.
10. R. Draves, "The Case for Run-Time Replaceable Kernel Modules," *Proceedings of the Fourth Workshop on Workstation Operating Systems*, October 1993, pp. 160-164.
11. D. R. Engler, S. K. Gupta, M. F. Kaashoek, "AVM: Application-Level Virtual Memory," *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, May 1994, pp. 72-77.
12. T. A. Funkhouser, *Database and Display Algorithms for Interactive Visualization of Architectural Models*, Ph.D. dissertation, University of California at Berkeley, 1993.
13. K. Gee, S. Murman, and L. Schiff, "Computation of F/A-18 Tail Buffet," *Journal of Aircraft*, Vol. 33, No. 6, Nov-Dec 1996, pp. 1181-1189.
14. K. Harty and D. R. Cheriton, "Application-Controlled Physical Memory Using External Page-Cache Management," *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1992, pp. 187-197.
15. J. Hultquist, personal communication, December 1996.
16. K. Hwang and F. Briggs, *Computer Architecture and Parallel Processing*, McGraw Hill, New York NY, 1984.
17. D. Jespersen and C. Levit, "Numerical Simulation of Flow Past a Tapered Cylinder," RNR Technical Report RNR-90-021, NASA Ames Research Center, October 1990.
18. G. Kiczales, J. Lamping, C. Maeda, D. Keppel, D. McNamee, "The Need for Customizable Operating Systems," Proceedings of the Fourth Workshop on Workstation Operating Systems, October 1993, pp. 165-169.
19. D. Lane, "UFAT: A Particle Tracer for Time-Dependent Flow Fields," *Proceedings of Visualization '94*, Washington DC, October 17-21, 1994, pp. 257-264.
20. C. H. Lee, M. C. Chen, and R. C Chang, "HiPEC: High Performance External Virtual Memory Caching," *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, November 1994, pp. 153-164.
21. D. McNamee and K. Armstrong, "Extending the Mach External Pager Interface to Accommodate User-Level Page Replacement Policies," *Proceedings of the USENIX Association Mach Workshop*, 1990, pp. 17-29.
22. National Space Science Data Center, *CDF User's Guide, Version 2.4*, NASA/Goddard Space Flight Center, February 1994.
23. U. Neumann, "Parallel Volume-Rendering Algorithm Performance on Mesh-Connected Multicomputers," *Proceedings of the 1993 Parallel Rendering Symposium*, San Jose CA, October 1993, pp. 97-104.
24. A. Silberschatz, J. Peterson, and P. Galvin, *Operating System Concepts*, Addison-Wesley, Reading MA, 1991.
25. D. Song and E. Golin, "Fine-Grain Visualization Algorithms in Dataflow Environments," *Proceedings of Visualization '93*, October 1993, pp. 126-133.
26. S. K. Ueng, K. Siborski, and K. L. Ma, "Out-of-Core Streamline Visualization on Large Unstructured Meshes," ICASE Report No. 97-22, Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, April 1997.
27. M. Young, *Exporting a User Interface to Memory Management from a Communication-Oriented Operating System*, Ph.D. dissertation, Carnegie Mellon University, November 1989.
28. P. Walatka and P. Buning, *PLOT3D User's Manual Version 3.6*, NASA Technical Memorandum 101067, NASA Ames Research Center, 1989.

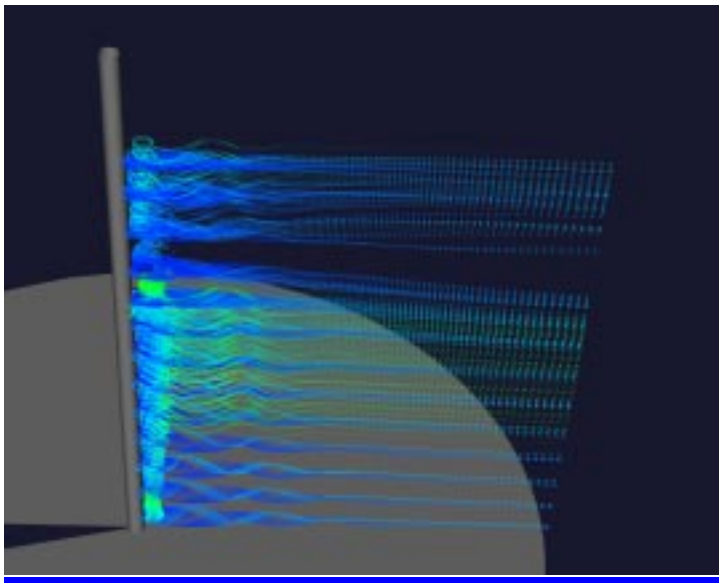

Plate 1. Tapered cylinder. Concatenated frames from an unsteady-flow particle trace simulation.

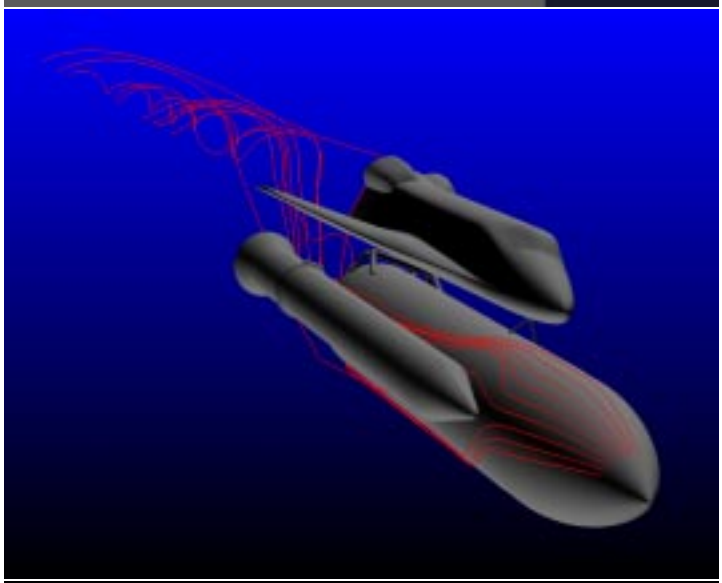

Plate 2. Shuttle. Single frame showing steady flow streamlines.

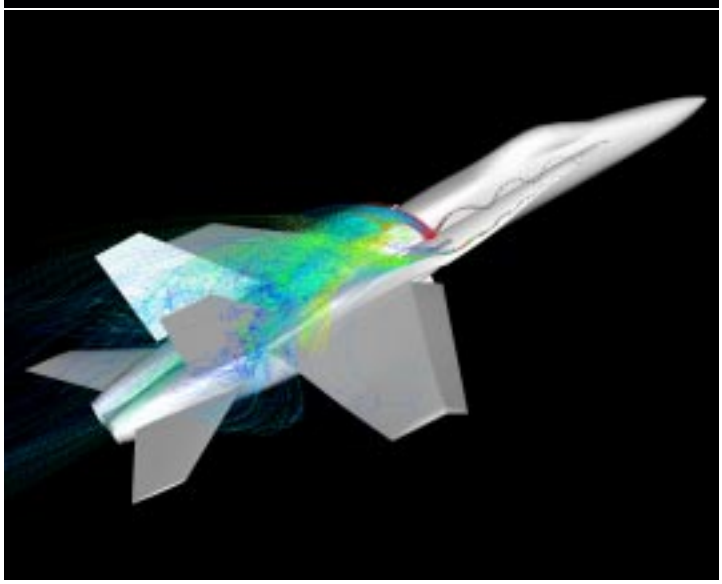

Plate 3. F18. Concatenated frames from an unsteady-flow particle trace simulation.