The Virtual Microscope

Ümit Çatalyürek, Michael D. Beynon, Chialin Chang, Tahsin Kurc, Alan Sussman, and Joel Saltz

***Abstract*—We present the design and implementation of the Virtual Microscope, a software system employing a client/server architecture to provide a realistic emulation of a high power light microscope. The system provides a form of completely digital telepathology, allowing simultaneous access to archived digital slide images by multiple clients. The main problem the system targets is storing and processing the extremely large quantities of data required to represent a collection of slides. The Virtual Microscope client software runs on the end user's PC or workstation, while database software for storing, retrieving and processing the microscope image data runs on a parallel computer or on a set of workstations at one or more potentially remote sites. We have designed and implemented two versions of the data server software. One implementation is a customization of a database system framework that is optimized for a tightly coupled parallel machine with attached local disks. The second implementation is component-based, and has been designed to accommodate access to and processing of data in a distributed, heterogeneous environment. We also have developed caching client software, implemented in Java, to achieve good response time and portability across different computer platforms. The performance results presented show that the Virtual Microscope systems scales well, so that many clients can be adequately serviced by an appropriately configured data server.**

***Index Terms*—Data caching, digital microscopy, grid computing, parallel computing, telepathology.**

I. Introduction

DESPITE numerous advances in the understanding of disease processes, most basic aspects of anatomic pathology have changed little over time. The pathologist supervises the gross dissection of tissue, which is fixed, dehydrated in organic solvents, embedded in paraffin, sectioned and stained. The tissue specimen is typically directly examined using a light microscope. The pathologist renders a diagnosis upon the microscopic examination of the tissue sections, and the glass slide and paraffin blocks are inevitably relegated to some cumbersome archive. A similar system is employed for cytopathology, but an added complication is that the slide is often unique and irreplaceable. Thus, the dissemination of case material for consultative, investigative or educational purposes remains laborious, and, to a large extent, pathologists only have access to locally available case material for comparison in difficult cases.

Over the past 10 years, there has been increasing interest in technologies that make it possible to examine specimens at a distance. There are currently two forms of telepathology imaging: static and dynamic [13], [30], [37]–[39]. In static-image telepathology, the referring pathologist captures a small set of digital images that are transmitted to the consultant. The consulting pathologist relies on the referring pathologist to select tissue fields. In the dynamic mode, live images of microscope slides are transmitted and visualized in real time. The dynamic form of telepathology can be carried out by a remotely controlled real microscope. The remote pathology consultant is abel to control the microscope stage and to select the image to be viewed. Advanced microscopes provide the functionality for selecting various color filters or applying different illumination modes. They also can allow the simultaneous viewing of a slide by multiple clients, although only one client can control the microscope. One main advantage of using a real microscope is that live specimens can be viewed in real time. A software system that allows access to digitized microscopy slides, on the other hand, can provide a cost-effective, complementary tool for dynamic telepathology. By simply emulating the usual behavior of a physical microscope, such a system can replace cabinets full of slides with a digital storage subsystem. Retrieving a slide then becomes a matter of accessing the slide database, without requiring physical access to the slide. As in a real microscope, it can provide simultaneous access to the slides by multiple users, who can access and individually manipulate the same slide or different slides at the same time. In addition, new software modules can be added to perform various types of additional processing, such as three dimensional image reconstruction from data found in multiple focal planes and on multiple microscope slides, image segmentation and pattern recognition to better characterize known malignancies, and content-based image retrieval, to find all slides with features similar to those in a sample slide [18], [40].

While the hardware for digitizing tissue samples and microscopy slides more effectively is rapidly becoming commercially available [24], the software support required to store, retrieve, and process digitized slides to provide interactive response times for the standard behavior of a physical microscope remains a challenging issue. A the basic level, the system should emulate the usual behavior of a physical microscope, including continuously moving the stage and changing magnification and focus. The processing for viewing a slide requires projecting

Manuscript received July 10, 2001; revised August 19, 2002 and September 13, 2002. This work was supported by the National Science Foundation under Grants #ACI-9619020 (UC Subcontract #10152408), #EIA-0121177, #ACI-0130437, and #ACI-9982087, the Office of Naval Research under Grant #N6600197C8534, Lawrence Livermore National Laboratory under Grant #B500288 and #B517095 (UC Subcontract #10184497), and the Department of Defense, Advanced Research Projects Agency, USAF, AFMC through Science Application International Corporation under Grant #F30602-00-C-0009 (SAIC Subcontract #4400025559).

Ü. Çatalyürek, T. Kurc, and J. Saltz are with the Department of Biomedical Informatics, The Ohio State University, Columbus, OH 43210 USA (e-mail: catalyurek.1@osu.edu; kurc.1@osu.edu; saltz.3@osu.edu).

M. D. Beynon was with the Department of Computer Science, University of Maryland, College Park, MD 20742 USA (e-mail: beynon@cs.umd.edu). He is currently at MIT Lincoln Laboratory, Cambridge, MA 02139 USA.

C. Chang and A. Sussman are with the Department of Computer Science, University of Maryland, College Park, MD 20742 USA (e-mail: chialin@cs.umd.edu; als@cs.umd.edu).

Digital Object Identifier 10.1109/TITB.2004.823952

high resolution data onto a grid of suitable resolution (governed by the desired magnification) and appropriately compositing pixels mapping onto a single grid point, to avoid introducing spurious artifacts into the displayed image.

The main difficulty in providing the basic functionality is storing and processing the extremely large quantities of data required to represent a large collection of slides. For example, with a digitizing microscope a single $200\times$ spot of a slide at a single depth of focus requires a resolution of 1000 by 1000 pixels. With a three-byte RGB color value per pixel, an image at that resolution produces a data size of 3 MB. To completely cover a slide of 3.5 by 2.5 cm requires a grid of about 50×70 such $200\times$ spots, resulting in an uncompressed file size of 10.5 GB. However, such as image captures only a single focal plane. Because histopathology involves interpretation of the three-dimensional (3-D) nature of tissue or individual cells, a single focal plane may not allow adequate characterization of the material. To acquire a slide at 5 focal planes increases the file size to 52.5 GB, and a higher power substantially increases the volume of these datasets with more spots and more focal planes. Storage needs are exacerbated by the fact that hospitals can generate many thousands of slides per year. For instance, at the Johns Hopkins Hospital the histology laboratory processes 420 000 routine, special-stain, and immunohistochemical slides per year. Clearly there is an enormous storage requirement. There are also the attendant difficulties in achieving rapid response time for various types of inquiries into the slide image database.

This paper describes the design and implementation of a complete software system, called the Virtual Microscope (VM), that implements a realistic digital emulation of a high power light microscope, through a client/server hardware and software architecture [2], [17]. The client software runs on an end user's PC or workstation, providing a graphical user interface (GUI) for viewing slides, while the database software for storing, retrieving and processing the microscope image data runs on a parallel computer or on a cluster of workstations at a potentially remote site. In terms of telepathology imaging, the Virtual Microscope can best be described as a form of completely digital telepathology. The contributions of this paper are as follows.

- We describe an implementation of the VM server using an object-oriented framework, called the Active Data Repository (ADR), for developing databases of multidimensional datasets on distributed memory parallel machines. Our previous work [11], [16] used VM as a motivating application scenario for the design of ADR. In this paper, we focus on the efficient implementation of VM using ADR. The Virtual Microscope implementation described in the prior work suffered from the overhead of extra function calls, resulting in about 85% slower execution than the original custom VM server implementation. The current implementation eliminates the extra function calls and achieves much better response times. The current ADR implementation of the VM server is only 6.6% slower than the original VM server [2], [17]. We also examine the effect on performance of partitioning a VM dataset into data chunks, and look at the scalability of the ADR implementation, when the number of clients and the number of processors are varied.
- We compare the performance of the ADR implementation of the VM server against a component-based implementation. We experimentally evaluate the two implementations so as to identify when it is beneficial to use the component-based implementation over the ADR implementation, and vice versa.
- We present the design and implementation of a client with data caching capabilities. Our experimental results show that data caching at the client improves client response time. It reduces contention among clients for scarce resources, such as bandwidth in a wide-area network and processing and I/O in the data server. Caching also improves client response time by reducing the amount of data requested from a (potentially remote) server.

In Section II, we provide an overview of the components of the Virtual Microscope system. Section III presents the implementation of the Virtual Microscope server using the Active Data Repository. An experimental evaluation of the implementation is also discussed in this section. The component-based implementation of the VM server and the performance comparison of the ADR implementation to the component-based implementation are presented in Section IV. We describe and experimentally evaluate the design and implementation of the caching client in Section V. Conclusions are given in Section VI.

II. Overall System Architecture

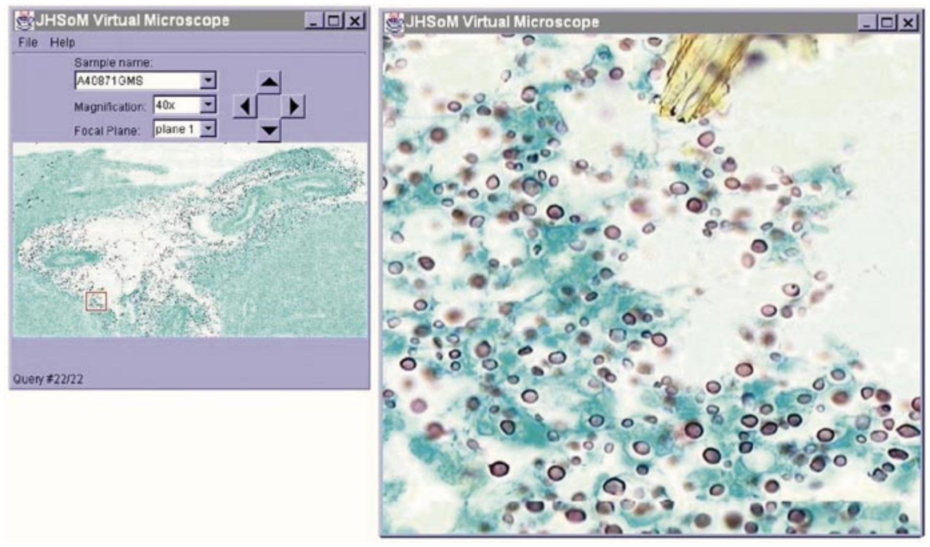
The basic functionality of the Virtual Microscope implements an accurate emulation of a high power light microscope. A number of operations must be supported to provide this functionality:

1) fast browsing through the slide to locate an area of interest;
2) local browsing to observe the region surrounding the current view;
3) changing magnification;
4) changing the focal plane.

The system design of the Virtual Microscope aims to support these four operations efficiently. The overall system employs a multitier software architecture with three main tiers; client, server frontend, and data server. The client is a graphical user interface that allows a user to perform the four basic actions, and generates requests to the server frontend as a result of user actions. The frontend interacts with clients and translates client requests into queries to the data server. The data server manages digitized slides, processes the queries and returns image data to the client.

A. Client Interface

The Internet-downloadable Java client program, shown in Fig. 1, provides a graphical user interface so that users can control browsing through slides by dragging and clicking the mouse. The current client is fully implemented in *Java 2* to achieve portability across different platforms. The client software is designed to run on an end user's PC or workstation. The client can run as a stand-alone application to be able to use the local disk of the client machine for caching. It can also be used as a helper application for an Internet browser. That is,



Sample name: A40871GMS
Magnification: 40x
Focal Plane: plane 1
Query #22/22

Fig. 1. The Virtual Microscope client.

a user viewing a web page conaining slide thumbnail images can start the Virtual Microscope client by simply clicking on a hyperlink. Upon starting up, the client program connects to the frontend, and, in a separate thread, listens for connections from the data server. The communication between the client program and the frontend and data server is done via TCP/IP sockets.

After selecting a slide, a client receives a *thumbnail image*, which is a low-resolution overview of the entire slide image, from the data server. A slide can be viewed at any of several available magnifications. In response to the user's actions, queries are generated by the client and sent to the server. The Virtual Microscope Java client consist of two windows:

1) The **display window** shows the selected portion of a slide at a selected magnification (the window on the right in Fig. 1).
2) The **control window** provides the standard operations supported by the Virtual Microscope as described previously (the window on the left in Fig. 1). The control panel has five subcomponents:
    - a *sample* selection box
    - a *magnification* selection box
    - a *focal plane* selection box
    - a *thumbnail* image, and
    - four *directional* buttons.

The thumbnail image in the control window presents a small, low magnification version of the entire slide and provides a user with two types of browsing operations. First, the user can locate the interesting portion of a slide rapidly by dragging the mouse on the small box (query box) inside the thumbnail window. Second, the user can move the microscope stage by small increments in one of the four directions (i.e., up, down, left, right) by clicking the corresponding directional button. The query box inside the window indicates the current portion of the image shown in the display panel.

Both the control window and the display window are resizable. When the user resizes the display window, the size of the query box inside the thumbnail window also changes accordingly. The display window is continuously updated while the user is panning through the image, either using actual image data cached at the client from previous queries (see Section V) or from the lower resolution thumbnail image. Once the user stops dragging and releases the mouse button, a query is generated and is satisfied either from the client cache or from the server, with the display area updated from the full image data at the desired resolution.

B. Server Frontend

The frontend interacts with clients and receives client requests, translates them into queries for the data server, and schedules them for processing by the data server. The frontend is a sequential program and runs on a workstation. Having a separate frontend has two main advantages. First, since clients can generate queries asynchronously, the existence of a frontend relieves the data server from being interrupted by the clients during processing of queries. Second, if a client is behind a firewall, the result of a query must be funneled through the frontend. In normal operation, the result is sent back to the client directly from the data server.

C. Data Server

The data server is the program responsible for efficiently serving image data. In order to produce an image, the data has to be read from disk and an image of the specified magnification must be reconstructed. Since the ultimate goal of the Virtual Microscope is to provide users with the illusion that they are using a physical microscope, the system must be able to support the standard functions of a physical microscope in software with a similar level of responsiveness and ease of use. These requirements present technical challenges in the design and implementation of the data server. The image database must provide low latency retrieval of large volumes of two dimensional image data (representing a portion of a focal plane of a given slide) from disk as well as efficient directory management for a large collection of slides.

As data from disks becomes available in memory, further processing is required to produce an image at the magnification level desired by the client. A query is processed by projecting high resolution data onto a grid of suitable resolution (governed by the magnification level requested by the client) and appropriately compositing pixels that map to a single grid point, to avoid introducing spurious artifacts into the displayed image. In order to achieve good performance, the server should be a scalable program. It should be designed to run on a parallel machine or on a cluster of workstations, with each node having several local disks. In addition, the server should be able to take advantage of asynchonous I/O operations and overlap the computation for a bolck with I/O for other blocks.

In the next section, we describe an implementation of the data server using an object-oriented framework, called the Active Data Repository (ADR), to address the above challenges.

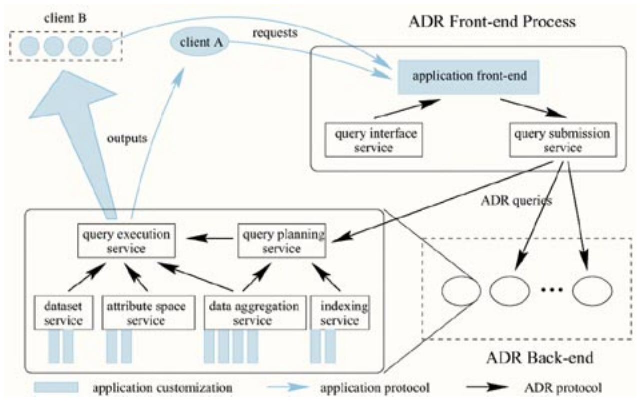


Fig. 2. An application suite implemented using ADR. The shaded bars represent functions added to ADR by the user as part of the customization process. Client A is a sequential program while client B is a parallel program.

III. Virtual Microscope Server Using the Active Data Repository

We have developed the Active Data Repository (ADR) [11], [16] to provide support for integrating application-specific processing with the storage and retrieval of multidimensional datasets on a parallel machine with a disk farm. In a multidimensional dataset, each data item is associated with a point in a multidimensional space. For instance, a digitized VM slide can be viewed as a 3-D dataset; each focal plane is a two-dimensional (2-D) image, and multiple focal planes constitute the third dimension. A reference to the data of interest is described by a *range query*, which is a multidimensional box defined in the underlying attribute space of the dataset. Only the data items whose associated points fall inside the multidimensional box are retrieved—an index (e.g., an R-tree [21]) can be used to quickly locate the data items to be retrieved. The main data processing steps consist of mapping the input data items to output data items, and aggregating all the input data items that map to the same output data item. An intermediate data structure, called an *accumulator*, can be used to hold intermediate results during processing.

A. Active Data Repository

The Active Data Repository consists of a set of modular services, implemented as a C++ class library, and a runtime system. Several of the services allow customization for user-defined processing. An application developer has to provide accumulator data structures for holding intermediate results, and functions that operate on *in-core* data to implement application-specific processing of *out-of-core* data. A unified interface is provided for customizing ADR services via C++ class inheritance and virtual functions. The runtime infrastructure supports common operations such as index creation and lookup, management of system memory, and scheduling of data retrieval and processing operations across a parallel machine. Multiple application-specific customizations of ADR services can co-exist in a single ADR instance, and the runtime system can manage multiple datasets simultaneously.

ADR provides support for implementing a *front-end process*, and a customized *back-end* (see Fig. 2). The front-end interacts with clients, translates client requests into queries and sends one or more queries to the parallel back-end. The back-end is responsible for storing datasets and carrying out application-specific processing of the data on the parallel machine. The customizable ADR services in the back-end include: 1) an *attribute space service* that manages the registration and use of user-defined mapping functions; 2) a *dataset service* that manages the datasets stored in the ADR back-end and provides utility functions for loading datasets into ADR; 3) an *indexing service* that manages various indices (default and user-provided) for the datasets stored in ADR; and 4) a *data aggregation service* that manages the user-provided functions to be used in aggregation operations, and functions to generate the final outputs. This service also encapsulates the data types of both the intermediate results (i.e., accumulator) used by those functions and the final output datasets.

1) Datasets in ADR: A dataset in ADR is stored as a set of data chunks, each of which consists of a subset of data items. A chunk is the unit of data retrieval in ADR. That is, a chunk is retrieved as a whole during processing. Retrieving data in chunks instead of as individual data items reduces I/O overheads (e.g., disk seek time), resulting in higher application level I/O band-width. As every data item is associated with a point in a multidimensional attribute space, every chunk is associated with a minimum bounding rectangle (MBR) that encompasses the coordinates of all the items in the chunk. The dataset is partitioned into data chunks by the application developer, and data chunks in a dataset can have different sizes. Since data is accessed through range queries, it is desirable to have data items that are close to each other in the multidimensional space placed in the same data chunk.

Data chunks are distributed across the disks in the system to fully utilize the aggregate storage space and disk bandwidth. In order to take advantage of the data access patterns exhibited by range queries, data chunks that are close to each other in the underlying attribute space should be assigned to different disks. By default, the ADR data loading service employs a Hilber curve-based declustering algorithm [15], [28] to distribute the chunks across the disks. Hilbert curve algorithms are fast and exhibit good clustering and declustering properties. Other declustering algorithms, such as those based on graph partitioning [29], can also be used by the application developer. Each chunk is assigned to a single disk, and is read and written only by the local processor to which the disk is attached. After data chunks are assigned to disks, a mulit-dimensional index is constructed using the MBRs of the chunks. The index on each processor is used to quickly locate the chunks with MBR's that intersect a given range query. An R-tree [21] implementation is provided as the default indexing method in ADR, but user-defined indexing methods can also be implemented.

2) Processing in ADR: The processing of a query in ADR is accomplished in two steps: a *query plan* is computed in the *query planning* step, and the actual data retrieval and processing is carried out in the *query execution* step according to the query plan.

Query planning is carried out in three phases: *index lookup*, *tiling* and *workload partitioning*. In the index lookup phase, indices associated with the datasets are used to identify all the chunks that intersect with the query. If the output/accumulator data structure is too large to fit entirely in memory, it is partitioned into *tiles* in the tiling phase. The ADR data aggregation service provides C++ base classes, which are customized by an application developer for tiling the accumulator data structure. Each tile contains a subset of the accumulator elements so that the total size of a tile is less than the amount of memory available for the accumulator. In the current ADR implementation, the workload partitioning step replicates the entire accumulator tile on each back-end processor, and each processor is responsible for processing local input data chunks. In the query execution step, the processing of an ouput tile is carried out according to the query plan. A tile is processed in four phases.

1) **Initialization**. Accumulator elements for the current tile are allocated space in memory and initialized in each processor.
2) **Local Reduction**. Each processor retrieves and processes data chunks stored on local disks. Data items in a data chunk are mapped to accumulator elements and aggregated using user-defined functions. Partial results are stored in the local copy of the accumulator tile on a processor.
3) **Global Combine**. Partial results computed in each processor in phase 2 are combined across the processors via inter-processor communication to compute final results for the accumulator.
4) **Output Handling**. The final output for the current tile is computed from the corresponding accumulator values computed in phase 3. The output is either sent back to a client or stored back into ADR.

A query iterates through these phases repeatedly until all tiles have been processed and the entire output has been computed. The output can be returned to the client from the backend nodes, either through a socket interface or via Meta-Chaos [14]. The socket interface is used for sequential clients, while the Meta-Chaos interface is mainly used for parallel clients.

Note that ADR assumes the order the input data items are processed does not affect the correctness of the result, i.e., aggregation operations are commutative and associative. Therefore, the runtime system can order the retrieval of input data chunks to minimize I/O overheads. Moreover, disk operations, network operations and processing are overlapped as much as possible during query processing. Overlap is achieved by maintaining explicit queues for each kind of operation (data retrieval, message sends and receives, data processing) and switching between queued operations as required. Pending asynchronous I/O and communication operations in the operation queues are polled and, upon their completion, new asynchronous operations are initiated when more work is required and memory buffer space is available. Data chunks are therefore retrieved and processed in a pipelined fashion. For portability reasons, the current ADR implementation uses the POSIX `lio_listio` interface for its nonblocking I/O operations, and MPI [33] as its underlying interprocessor communication layer.

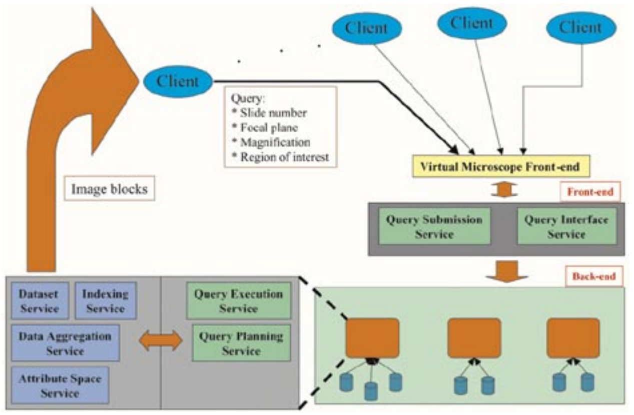


Fig. 3. Virtual Microscope customization of the Active Data Repository.

The backend can execute multiple queries concurrently. Each query is assigned its own workspace (e.g., memory for the accumulator data structure). The runtime system switches between queries to issue I/O and communication operations, and handles the computation for a query when the corresponding I/O and communication operations complete.

B. The Virtual Microscope Implementation Using ADR

We now discuss how digitized microscopy images are stored for the efficient processing of VM queries, and describe the VM-specific customization of the ADR services (see Fig. 3).

1) Storing Digitized Images: Managing extremely large quantities of data is the major problem in the design and implementation of the Virtual Microscope. The large volume of image data requires effective use of a large number of disk units, which in turn requires effective placement of multidimensional data sets (each slide consisting of multiple 2-D focal planes) onto a large disk farm to maximize disk access parallelism and minimize disk access latency.

We focus on *parallelism* and *locality* of data retrieval from secondary storage. Disk access parallelism reduces the volume of data retrieved from individual disk units, thereby minimizing overall query processing time. On the other hand, disk access locality affects the amount of time spent to locate the data objects on a single disk (i.e., disk seek and latency).

The digitized image from a slide is essentially a three dimensional data set, because each slide may consist of multiple focal planes. In other words, each digitized slide consists of several 2-D images stacked on top of one another. However, the portion of the entire image that must be retrieved to provide a view into the slide for any given set of microscope parameters (area of interest, magnification and focal plane) is two dimensional. Therefore, to optimize performance, each 2-D image (a focal plane) should be an independent unit for the data declustering algorithm to maximize disk parallelism, whereas the entire 3-D data set (a set of focal planes) should be considered together by the data clustering algorithm to improve data locality on each disk (changing focal planes does not change the area of interest within a plane). If each focal plane of the slide is stored as a single image, it constitutes a very large image. For example, the size of the tissue slides in our archive varies from 10 K $\times$ 10 K pixels to 60 K $\times$ 60 K pixels. In other words, these images require from 300 MB to 10.8 GB of storage. While current storage technology provides adequate capacity to store these large image, in order to process the images efficiently they must be partitioned into blocks (or *tiles*). If the whole image must be processed, the size chosen for the tiles should depend on the processing require for each tile and the processing power and I/O capabilities of the server hardware. Even if the processing that will be carried out on each tile does not require very much computation, storing very large tiles may reduce overall system performance if disk I/O becomes a bottleneck. In the Virtual Microscope system, the VM data server does not process the whole image for each query. Most queries require processing only a small portion of the image. Hence, the size of the tiles must be big enough to efficiently use the disk subsystem, but not so big that too much unneeded data is retrieved and processed. The shape of each tile can also affect the amount of data retrieved from disk. Tiles that are elongated in one dimension are likely to result in retrieval of much unneeded data. For example, if a slide is partitioned in the x-dimension into vertical strips, a query that spans the entire x-dimension, but only a small portion of the y-dimension, will, will cause the data for the entire slide to be read from disk. Thus, in the VM implementation, we partition a slide into tiles that are close to square in shape.

The images should also be stored in a compressed form. In the current implementation we have selected JPEG compression as the default method because of the availability of fast and stable compression/decompression libraries [22]. As we have pointed out previously [2], wavelet-based image compression appears to be the most appropriate technique for the Virtual Microscope system. However, we have tested two public domain wavelet compression libraries that implement that JPEG-2000 standard [10], [23], [26] and found that they are about 10 to 20 times slower for decompressing microscope images than the public domain implementation of the JPEG library from the Independent JPEG Group [22].

An image tile is used as the unit of data storage and retrieval in the Virtual Microscope customization of ADR. That is, a tile and its associated metadata (position of the tile in the whole image and its size) are stored as a single chunk in an ADR data file. The chunks are distriubuted across the system disks using a Hilbert curve-based declustering algorithm (see Section III-A1).

2) Customization of ADR Services: ADR provides an algorithm-independent interface that is used by the data aggregation service. During the processing of a query, the server process finds the image blocks that intersect the query region, and reads them from disks. A retrieved image block is first *decompressed*, since image blocks are stored on disks in a compressed format to reduce storage requirements. Then the block is *clipped* to the query region. Finally, each clipped block is *subsampled* to achieve the magnification (*zoom*) level specified by the query. The aggregation service is customized by implementing these four operations in a virtual method that is called by the ADR runtime system when a data chunk is available in memory. New image processing functionality for the Virtual Microscope system can be added by implementing new aggregation functions.

In the implementation of the indexing service customization, we have exploited the fact that the image tiles are nonoverlapping and that the slides are fully rectangular images without holes. The tiles are numbered in row-major order; hence, given the location of the tile the data chunk that contains the tile in JPEG format can be determined very quickly. If the system needs to store datasets with holes in the images, or images that are not rectangular, the default R-tree indexing method in ADR can be employed.

The output of a VM query is a 2-D image produced by clipping the input image tiles to the query window and subsampling the clipped tiles to get the desired magnification. A 2-D array can be used as an accumulator to hold pixels from the clipped and subsampled input image tiles. In that case, the 2-D array is replicated in each processor, and the final image is computed in the global combine and output handling steps of the query execution phase in ADR. However, the clipping and subsampling of an image tile can be done independently of other image tiles, and image tiles contain disjoint subsets of the pixels in the entire slide. As a result, each processor needs to allocate only the portions of the output image that are computed by processing the local image tiles. By *sparsely* allocating the accumulator in this way, we can reduce the aggregate system memory required for the accumulator, and do not need to perform the ADR global combine step. The resulting image blocks are stored in memory during the local aggregation step, and directly sent to the client in the output handling step. The client assembles and displays the image blocks from the data server to form the query output.

C. Experimental Results

In this section we present experimental performance results for the ADR version of the Virtual Microscope server running on a Linux PC cluster. The PC cluster consists of one front-end node and five processing nodes, with a total of 800 GB of disk storage. Each processing node has an 800 MHz Pentium III CPU, 128 MB main memory, and two 5400 RPM Maxtor 80 GB EIDE disks. The processing nodes are interconnected via 100 Mb/s switched Ethernet. The front-end node is also connected to the same switch.

We have used the driver program described in [5] to emulate the behavior of mulitple simultaneous end users (clients). The implementation of the client driver is based on a workload model that has been statistically generated from traces collected from real experienced users. Interesting regions are modeled as points in the slide, and provided as an input file to the driver program. When a user pans *near* an interesting region, there is a high probability a request will be generated. The driver adds noise to requests to avoid multiple clients asking for the same region. In addition, the driver avoids having all the clients scan the slide in the same manner. The slide is swept through in either

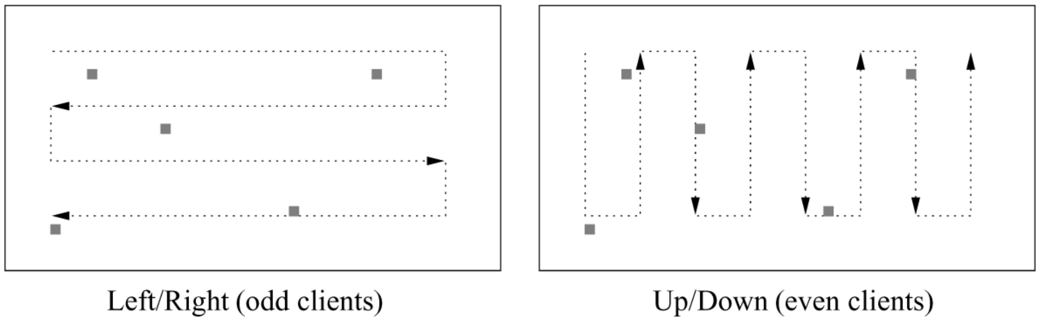


Fig. 4. Sweeping patterns over interesting points.

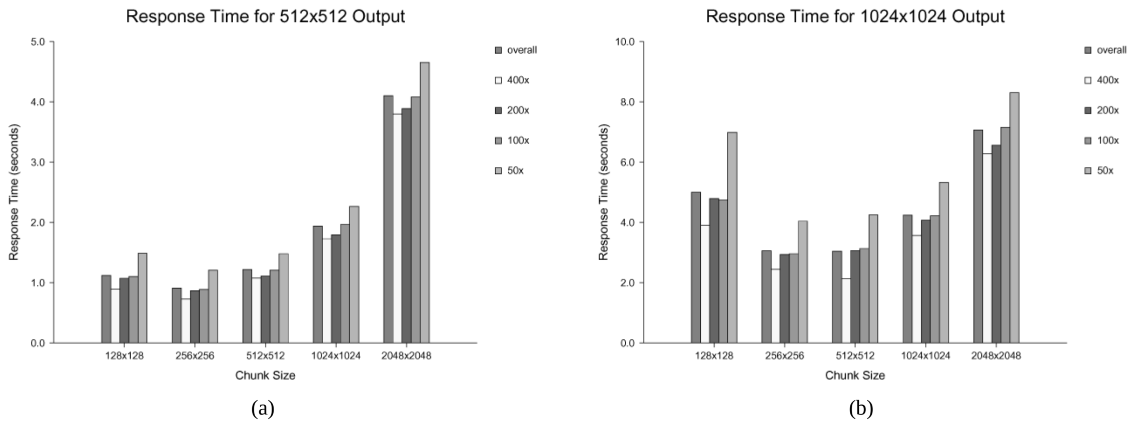


Fig. 5. Performance results for the Virtual Microscope ADR server running on 5 processors for varying image chunk sizes. Average response times of the server for the queries to produce an output image.

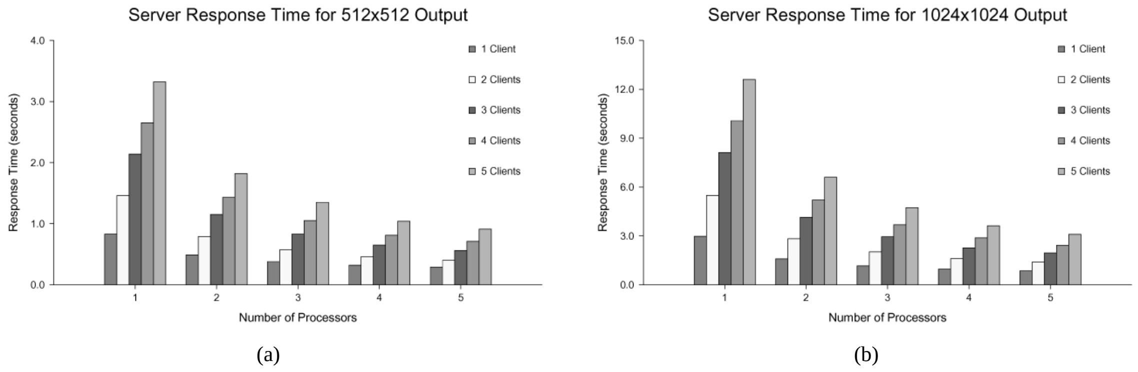
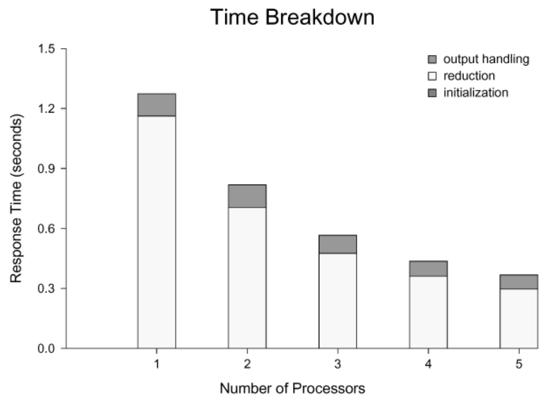


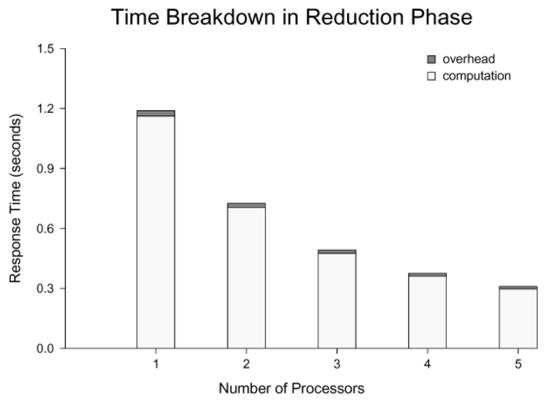
Fig. 6. Performance figures for Virtual Microscope ADR server on varying number of processors for generating the output image. In this experiment, each client submits 100 queries to the server.

an up-down fashion or a left-right fashion, as shown in Fig. 4, as observed from real users. For the experiments presented in this section, each client generates 100 queries. The generated query set contains queries at different resolutions, hence some of the queries (those at lower resolutions) require processing more data at the VM data server, since the data is stored at the highest resolution. For example, a query at 50$\times$ magnification requires processing 64 times more data than a query requesting an output at 400$\times$ magnification. The response times that are shown in Figs. 5–7 are the average response times for a single query. In the experiments we have used a tissue slide of size $32\,336 \times 27\,840$ pixels.

The performance results for the VM data server using different chunk sizes are displayed in Fig. 5. In this figure, the 400$\times$, 200$\times$, 100$\times$ and 50$\times$ bars show the average respone times of the VM data server to the queries at different resolutions, where 400$\times$ is the highest resolution stored in the VM data server. The *overall* bar displays the average response times of the VM system to the set of queries at all resolutions. As seen in Fig. 5(a), chunk size 256×256 produces the best response time at each resolution, and therefore for the overall average for queries that request a 512×512 output image. Both 128×128 and 512×512 chunk sizes result in response times that are approximately 33% higher. Increasing the chunk size decreases system performance because with too large a chunk size all of the processing nodes in the VM data server cannot be efficiently utilized, especially for queries requesting a relatively small output image. As chunk size increases, the number



(a)



(b)

Fig. 7. Breakdown of query execution time into phases.

of chunks that intersect with a fixed size user decreases. For example, with chunk size 2048×2048 a query requesting an output of size 512×512 at the highest resolution intersects with either 1, or 2 or 4 chunks. It is highly probable that most such queries will intersect only 1 chunk because of the large chunk size. In that case , four processors of the five-processor VM data server will be idle. For queries that request a 1024×1024 output image (see Fig. 5(b)), the best response time is achieved by a chunk size of either 256×256 or 512×512. Again, increasing the chunk size decreases performance. Using a chunk size that is too small also decreases overall system performance because reading chunks from disk becomes a bottleneck, producing too many small disk I/O requests in the data server. Since a 256×256 chunk size gave the best response times for queries requesting both 512×512 and 1024×1024 output images, we have selected 256×256 as the default chunk size for the remaining experiment in this section.

Fig. 6 displays the average response times for queries that request 512×512 and 1024×1024 output images, with queries generated by multiple concurrently running clients. Each client is an instance of the driver program and generates 100 queries. The generated query set contains queries at different resolutions. The response times that are shown in these figures are the average response times for a single query. As is seen in Figs. 6(a) and (b), the performance of the ADR version of the VM server scales very well as both the number of clients and the output size increase. For example, for queries that request a 512×512 output image and with 5 clients, the speedup for five processors is 3.6 compared to a one processor server, whereas the speedup is somewhat worse (3.0) for one client. For queries that request a 1024×1024 output image, the speedup for one client with a five processors is 3.5, and for five clients the speedup is 4.1.

Fig. 7 displays a breakdown of the execution times of the ADR version of the VM data server. As described in Section III-A2, query execution in ADR has four phases: initialization, local reduction, global combine and output handling. In the VM customization, initialization only allocates space for the accumulator, which holds decompressed and clipped image chunks. Decompression, clipping and subsampling are done in the local reduction phase. In the VM data server there is no need for a global combine, since the VM client stitches together the image chunks received from the data server. In the output handling phase, clipped and subsampled images are compressed to reduce network traffic and sent to the VM client. As is seen in Fig. 7(a), most of the server time is spent in the local reduction phase and a small fraction is spent in the output handling phase. Initialization time is so small that it is not even visible in the figure. Fig. 7(b) displays a time breakdown for the reduction phase. In this figure, computation corresponds to the execution time of JPEG decompression, clipping and subsampling, after a data chunk is retrieved from disk and is available in memory. The remaining time in the reduction phase is denoted by overhead. As was discussed in Section III.A.2, data chunks are retrieved from disks via nonblocking I/O functions to overlap I/O with computation, and the ADR runtime system manages buffer space and performs scheduling of I/O, network, and computation operations. Thus, in Fig. 7(b), *overhead* includes nonoverlapped I/O and other overheads incurred by the runtime system. The overhead is very small compared to the computation time; it is approximately 3% of the computation time, on average. Our results show that most of the I/O is overlapped with computation, and very little overhead is incurred by the runtime system.

IV. The Virtual Microscope Server in a Distributed Computing Environment

The ADR implementation of the Virtual Microscope aims to provide a scalable, portable, and customizable data server optimized for disk-based datasets on a tightly-coupled, homogeneous parallel machine. Advances in networking, computing, and storage technologies are rapidly making it possible to effectively use a networked collection of storage and computing systems. Although a networked collection of storage and computing systems offers a powerful and flexible environment, it requires distributed access and processing of data in a heterogeneous environment. The heterogeneity can arise for several reasons: 1) cpu/disk/memory resources are nonuniform, perhaps caused by multiple purchases of equipment over time; 2) space availability can dictate sub-optimal placement of the application dataset on disks, causing nonuniform data retrieval costs; 3) these can be unexpected or nonuniform application access patterns, including data subsetting, into the dataset; 4) shared resources, such as cluster nodes, can result in varying resource availability. To attain good performance when heterogeneity is present, applications should be flexible. Moreover, they should be optimized in the use of resources and be able to adapt to changes in their availability.

Consider a scientist who wants to compare properties of a 3-D reconstructed view of a dataset, recently generated by a digitizing microscope at a collaborating institution, with the properties of a large collection of refrence datasets. The 3-D reconstruction operation involves retrieving portions of 2-D slides (or focal planes) from the regions in question, and then performing feature recognition and interpolating between the slices to extract the important 3-D features. A description of these features and the associated properties are then compared against a database of known features, and some appropriate similarity measure is computed. The final result is the set of reference features found that are close in some way to those found in the new raw dataset, along with the corresponding view renderings to visualize.

In this scenario, the required resources (new raw dataset, reference database, and the scientist) can all be at locations distributed across a wide-area network. The storage requirement for such a collection is enormous; the size of the entire collection generated in a year in one hospital can reach up to several hundred terabytes. Since the reference dataset is very large and can be useful to many users, it is likely to be stored in an image library in one or more archival storage systems. Moreover, the archival storage systems may be located at different departments in a hospital or at different hospitals. The new raw dataset would be stored at the site where the slides were digitized. If the hosts containing the data are high capacity archival systems that do not allow the execution of the 3-D reconstruction code (or make it prohibitively expensive), it becomes unclear how to structure the application for efficient execution. Ideally, we would like to execute portions of the application at strategic points in the available collection of machines. For example, if the portion of code that performs the range select on the new raw dataset could be run on the host where the data lives, the amount of data to be transmitted over the wide-area network would be reduced. A computation farm could be an ideal location for the feature recognition and 3-D reconstruction due to the parallelism inherent in the codes.

There is a large body of research on building computational grids and providing support for enabling execution of applications in a wide-area environment [12], [19]. There is also hardware and software research on archival storage systems, including distributed parallel storage systems [27], file systems [35], high-performance I/O systems [36] and remote I/O [34]. However, providing support for efficient subsetting and processing of very large scientific datasets stored in archival storage systems in a distributed environment remains a challenging research issue. We have developed a middleware infrastructure, called DataCutter [4], [6], [8], that enables processing of scientific datasets stored in archival storage systems in a distributed, heterogeneous environment. In this section we describe an implementation of the Virtual Microscope server using the DataCutter infrastructure. We compare the DataCutter implementation of VM to the Active Data Repository implementation in a heterogeneous environment.

A. DataCutter

DataCutter provides a set of core services, namely an indexing service and a filtering service, on top of which more application specific services can be implemented. The indexing service provides support for accessing subsets of datasets via multidimensional range queries. To ensure scalability to very large datasets, DataCutter uses a multilevel hierarchical indexing scheme, implemented atop the R-tree index method [3]. The filtering service supports the filter-stream programming framework for executing application-specific processing as a set of components, called *filters*, in a distributed environment. Processing, network and data copying overheads are minimized by the ability to place filters on different hosts. The filtering service can be used to instantiate and execute collections of filters on various hosts, including networks of workstations and SMP clusters. The filtering service is designed to allow many filters to carry out resource constrained, pipelined communication and pipelined processing of data. DataCutter allows users to define multiple linked filters as well as sets of concurrent filter instances that are used collectively to perform computation; work can be directed to any running instance. DataCutter can be used to support data subsetting and user-defined filtering of large multidimensional datasets in a distributed environment. It can also be used to support the generation of new data products that can be subsequently visualized or stored.

1) Multilevel Indexing for Subsetting Very Large Datasets: We assume that a scientific dataset consists of a set of data files and a set of index files. Data files contain the data elements of a dataset; data files can be distributed across multiple storage systems. Each data file is viewed as consisting of a set of *data chunks*, as in ADR. Efficient spatial data structures have been developed for indexing and accessing multidimensional datasets, such as R-trees and their variants [3]. However, storing very large datasets may result in a large set of data files, each of which may itself be very large. Therefore a single index for an entire dataset could be very large. Thus, it may be expensive, both in terms of memory space and CPU cycles, to manage the index, and to perform a search to find intersecting data chunks using a single index file. Assigning an index file for each data file in a dataset could also be expensive because it is then necessary to access all the index files for a given search. To alleviate some of these problems, we have developed a multilevel hierarchical indexing scheme implemented via *summary index files* and *detailed index files*. The elements of a summary index file associate metadata (i.e., an MBR) with one or more data chunks and/or detailed index files. Detailed index file entries themselves specify multiple data chunks. Each detailed index file is associated with some set of data files, and stores the index and metadata for all data chunks in those data files. There are no restrictions on which data files are associated with a particular detailed index file for a dataset. Data files can be organized in an application-specific way into logical groups, and each group can be associated with a detailed index file for better performance. R-trees are used as the indexing method for summary and detailed index files.

2) Processing of Data: Filters and Streams: Recent research on programming models for developing applications in the Grid has converged on the use of component-based models [1], [20], [25], [31], [32], in which an application is composed of multiple interacting computational objects. In the DataCutter project, we have developed a framework, called filter-stream programming, for developing data intensive applications in a distributed environment. The filter-stream programming model represents processing components of a data-intensive application as a set of filters, which are designed to be efficient in their use of resources. Data exchange between any two filters is described via streams, which are uni-directional pipes that deliver data in fixed size buffers. The basic idea behind the use of the filter-stream programming model is to some-what constrain the behavior of a generic message passing application, so that the application can expose information that is useful for improving performance in several ways. Filters are location-independent, because stream *names* are used to specify filter to filter connectivity, rather than endpoint location on a specific host. This allows the placement of filters on different hosts in a distributed environment. Therefore, processing, network and data copying overheads can be minimized by the ability to place filters on different platforms.

A *filter* is a user-defined object with methods to carry out application-specific processing on data. A filter is specified by the application code to execute, and the layout of input and output streams it will use. Currently, filter code is expressed using a C++ language binding by subclassing a filter base class. This provides a well-defined interface between the filter code and the filtering service. The interface for filters consists of an initialization function, a processing function, and a finalization function.

```
class ApplicationFilter: public DC_Filter_Base_t {
    public:
        int init(int argc, char *argv[]) {...};
        int process (stream_t st[]) {...};
        int finalize(void) {...};
}
```

A *stream* is an abstraction used for all filter communication, and specifies how filters are logically connected. It provides the means of uni-directional data flow between two filters, from up-stream filter to downstream filter. Bi-directional data exchange is achieved by creating two streams in opposite directions. All transfers to and from streams are through a buffer abstraction. A buffer represents a contiguous memory region containing useful data. Streams transfer data in fixed size buffers. The size of a buffer is determined in the **init** call; a filter discloses a minimum and an optional maximum value for each of its streams. The actual size of the buffer allocated by the filtering service is guaranteed to be at least the minimum value. The optional maximum value is preferred buffer size hint to the filtering service. The size of the data in a buffer can be smaller than the size of the buffer. Therefore, the buffer contains a pointer to the start, the length of the portion containing useful data, and the maximum size of the buffer. In the current prototype implementation we use TCP for stream communication, but any point-to-point communication library could be added.

DataCutter provides several degrees of flexibility to improve application performance [7]–[9]. The choice of placement represents an important degree of freedom in affecting application performance by placing filters with affinity to data sources near the sources, minimizing communication volume on slow links, and placing filters to deal with heterogeneity. Parallelism in executing application-defined queries via group instances and parallel filters is another degree of freedom. Group instances enable *inter-query* parallelism by concurrent instances of filter groups, because multiple queries can be processed concurrently by different group instances. Parallel filters, on the other hand, allow a finer level of *intra-query* parallelism via multiple copies of a single filter within a single filter group.

A filter group is a set of running filters that are logically related and are used together to perform a computation. Multiple concurrent running instances of any number of filter groups is supported by the DataCutter runtime system. Each filter within a filter group is executed in a separate POSIX thread context, which allows for concurrent execution. Work can be appended to any running group instance and is handled in first-in–first-out (FIFO) order by an instance. There is no ordering between work appended to concurrent group instances.

If an application implemented using filters involves pipelined processing of data, the performance of the application depends on how well the stages of pipeline are balanced in terms of relative processing time of the stages and the ratio of communication cost between two stages to the computation cost of each stage. The performance penalty that is observed in an unbalanced pipeline can be addressed by using what we refer to as *transparent copies*, where the filter is unaware of the concurrent filter replication. We define a *copy set* to be all transparent copies of a given filter that are executing on a particular host. The DataCutter filtering service maintains the illusion of a single logical point-to-point stream for communication between a logical producer and a logical consumer in the filter group. When this logical producer and/or logical consumer has transparent copies, the filter service must decide for each producer which consumer to send a buffers to. Each copy set shares a single buffer queue, so there is perfect demand-based balance within a single host. For distribution between copy sets (different hosts), we have investigated several policies: 1) Round robin distribution of buffers among copy sets, 2) Round robin among copy sets based on the number of copies on that host, and 3) a Demand Driven sliding window mechanism. The Demand Driven policy is designed to send buffer to the filter that will result in the fastest processing. To approximate this, we instead send it to the filter that is showing recent good performance. When a consumer filter processes a data buffer received from a producer, it sends back an acknowledgment message to the producer that indicates the buffer is now being processed. The producer chooses the consumer filter with the fewest unacknowledged buffer to send a data buffer. The effect is to direct more buffer to faster consumers, with a cost of extra acknowledgment message traffic.

B. *The Virtual Microscope Server Using DataCutter*

The filter decomposition used for the Virtual Microscope system is shown in Fig. 8. This filter pipeline structure is natural for query-response applications. The figure depicts the main dataflow path of image data through the system. The

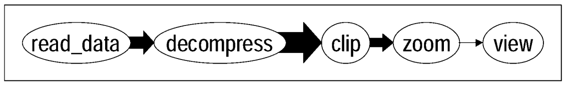


Fig. 8. Virtual Microscope decomposition.

thickness of the stream arrows indicate the relative volume of data that flows on the different streams. In this implementation each of the main processing steps in the server is a filter.

- **read_data (R):** Full-resolution data chunks that intersect the query region are read from disk, and written to the output stream.
- **decompress (D):** Image blocks are read individually from the input stream. Each block is decompressed using JPEG decompression and converted into three byte RGB format. The image block is then written to the output stream.
- **clip (C):** Uncompressed image blocks are read from the input stream. Portions of the block that lie outside the query region are removed, and the clipped image block is written to the output stream.
- **zoom (Z):** Image blocks are read from the input stream, subsampled to achieve the magnification requested in the query, and then written to the output stream.
- **view (V):** Image blocks are received for a given query, collected into a single reply, and sent to the client using the standard Virtual Microscope client/server protocol.

C. Experimental Results

1) Serving Digitized Microscopy Images From an Archival Storage System: We have implemented a simple data server for digitized microscopy images, stored in the IBM HPSS archival storage system at the University of Maryland [4], [6]. The HPSS setup has 10 terabytes (1 TB is 1000 GB) of tape storage space, 500 GB of disk cache, and is accessed through a 10-node IBM SP multicomputer. One node of the SP is used to run the filter that carries out index lookup, and the client was run on a SUN workstation connected to the SP node through the department Ethernet. The Virtual Microscope client trace driver was again used to drive the experiments. The driver was always executed on the same host as the *view* filter, which is referred to as the client host. The server host is where the *read_data* filter is run, which is the machine containing the dataset.

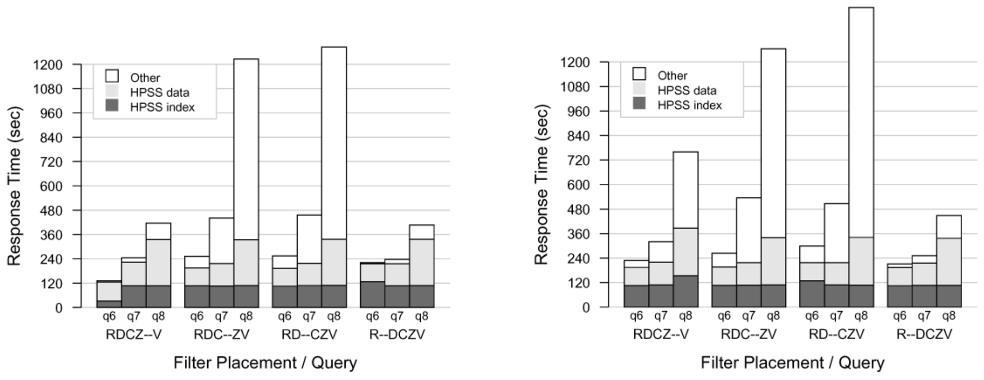
We experimented with different placements to the *read_data* (R), *decompress* (D), *clip* (C), *zoom* (Z), and *view* (V) filters by running some of the filters (and the filtering service) on the same SP node where the indexing service is executed, as well as on the SUN workstation where the client is run. In the next set of experiments (Fig. 9), we consider varying the server load. In these experiments, we used a scaled version of a VM dataset. The scaled dataset in 250 GB compressed (5.7 TB uncompressed), and corresponds to a 2-D image with 1.4 M $\times$ 1.4 M RGB pixels. The image is regularly partitioned in to data chunks and stored in 1024 files on the HPSS. In all experiments, we use a subsampling factor of 8. The execution times are response times seen by the visualization client averaged over three repeated runs for three queries, $q6, q7$, and $q8$. $q6$ covers 5×5 chunks of the image, $q7$ and $q8$ cover 4 times and 16 times the area covered by $q6$, respectively.

Fig. 9(a) and (b) shows query execution times when the server load is the same as the client load and when the server load is doubled. The different loads were emulated by artificially slowing down the set of filters running on the server host such that the total running time was increased. For example, the *zoom* filter runs twice as long in the 2$\times$ case because the filter is delayed. As is seen from the figures, running the filters at the client (R-DCZV) achieves better performance than running them at the server (RDCZ-V) as server load increases (or the client host becomes relatively faster). This result is not unexpected, but the experiment quantifies the effect for this particular configuration. The use of a different client to server network, or hosts with different relative speeds would significantly change the observed trends and tradeoff points.

2) A Comparison of ADR and DataCutter Servers: We now present an experimental comparison of the ADR and DataCutter implementations of the Virtual Microscope server. The experiments were carried out using a PC cluster and an SMP machine, both running Linux, at University of Maryland. The PC cluster has single-processor nodes, interconnected via Switched Fast Ethernet. Each node has a Pentium III 650 MHz CPU, 128 MB of main memory, and two 75 GB IDE disks. The SMP machine has 8 Pentium III 550 MHz processors, 4 GB of main memory, and 18 GB of disk space. In these experiments we used the same tissue slide that was used for the experiments in Section III-C. The slide consists of a single focal plane of $32\,336 \times 27\,840$ pixels. We used the client emulation driver program [5] to generate queries to the data server.

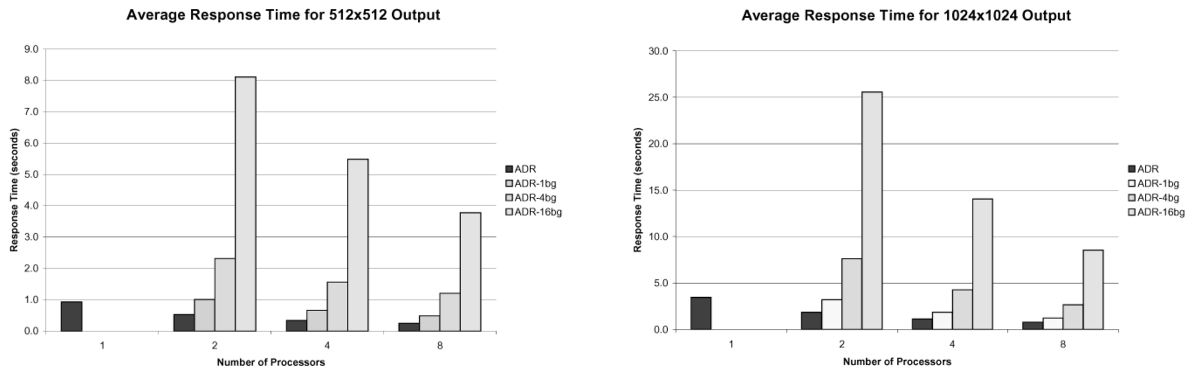
In the first set of experiments, we measure the effect of varying background load on some of the server nodes on the performance of the ADR server. Fig. 10 shows the average response time achieved by the ADR server. The response time was measured in the client driver program, which also performs the final stitching of partial images received from the server backend nodes. Each bar in the graphs represents the average response time for 100 queries. In these experiments, the input tissue slide was partitioned into chunks of 256×256 pixels, and the chunks were distributed across the nodes for each machine configuration so that each node has the same number of data chunks. For the experiments, background load was added to half of the nodes in each configuration by executing a user level job that consumes CPU time, at the same priority as the filter code. For the machine configuration with one processor, no background job was run. For 2-, 4-, and 8-processor configurations, we ran 1, 4, and 16 background jobs, denoted by *1bg*, *4bg*, and 16*bg* in the figures, on half of the processors in each configuration. As is seen from Fig. 10, the performance of the ADR server degrades significantly as background load increases. As was discussed in Section III, each backend node in the ADR server processes only the data stored on its local disk. As more background jobs are executed on some of the nodes, those nodes spend more time to process data chunks, even though a nearly equal number of data chunks are distributed to each node.

Figs. 11 and 12 show the performance of the ADR and DataCutter implementation of the VM server for queries



(a) (b)

Fig. 9. Execution time of queries under varying server load. R, D, C, Z, V denote the filters *read_data, decompress, clip, zoom*, and *view* respectively. The placement of the filters at the server and client is denoted by {server}—{client}. HPSS index is the index lookup time, HPPS data and Other are the the sum of the execution time for searching segments that intersect a query, and for processing the retrieved data via filters. The subsampling factor for queries is 8.

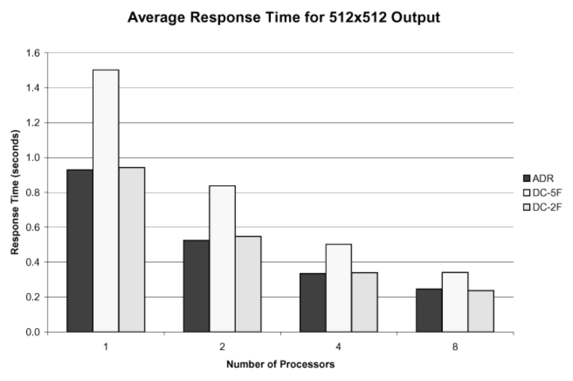


(a) (b)

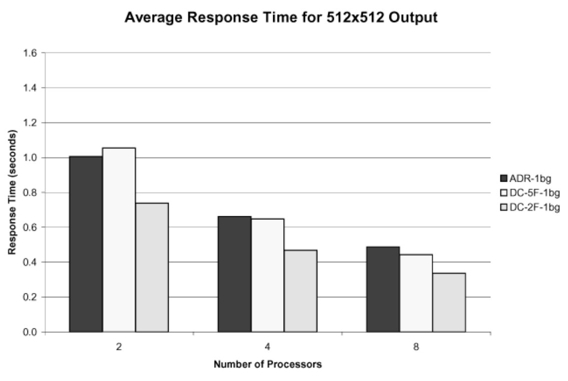
Fig. 10. Average response time of the ADR server, when the number of back-ground jobs and the number of processors are varied.

that produce 512×512 and 1024×1024 output images, respectively, when background load is varied. In these experiments, we used two different filter configuration for the DataCutter-based VM server: *DC-5F* denotes the data server with five separate filters, namely **R**ead, **D**ecompress, **C**lip, **Z**oom, and **V**iew. *DC-2F*, on the other hand, is the VM server implemented using two filters. In this configuration there is one read filter and the decompress, clip, zoom, and view operations are combined into a single filter. For the VM server versions implemented using DataCutter, one transparent copy of each filter was executed on each of the nodes in the system. We used the demand-driven sliding windows mechanism for on-the-fly distribution of data buffers between copy sets on different hosts (see Section IV-A2). As is seen from the figures, without background load the five-filter version of the data server, *DC-5F*, is slower than the ADR version. However, the two-filter version, *DC-2F*, performs as well as the ADR server. For *DC-5F*, data buffers between each stage of the pipeline are distributed using the demand-driven sliding windows mechanism. Although this configuration may result in better pipelining of processing and better load balance among transparent copies, the volume of data communication will be higher than that of both the ADR vesion and the *DC-2F* configuration. In this case, the extra communication overhead due to additional stages in the pipeline offsets the additional pipelining and load balance achieved. The response time of the ADR server discernibly increases as the number of background jobs rises. Background load also slows down the DataCutter version of the VM server. However, its effect is much less than that on the ADR server. For example, *DC-5F* is slower than ADR when there are no background jobs, but it becomes faster with even 1 background job on 8-processor configurations. With 4 background jobs, both DataCutter implementations. *DC-5F* and *DC-2F*, run faster than ADR implementation. As seen in the figure, the performance improvement of DataCutter implementations increases with the increasing number of background jobs. This is because of the dynamic distribution of data buffers among the transparent copies of a filter resulting from the demand driven scheme. When the load on a node increases, the data buffers from the read filter running on that node are sent by the rutime system to other nodes that are less loaded.

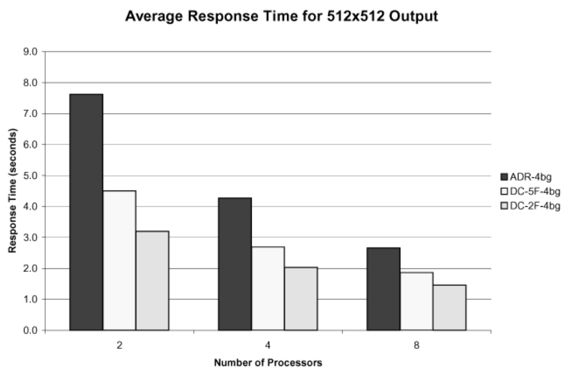
Fig. 13 shows average response times using the ADR and DataCutter implementations of the VM server on the 8-processor SMP machine, when the number of clients is varied. In this set of experiments, we ran 8 backend processes for the ADR server. Each client generates 100 queries and waits for the completion of a query before submitting a new query. As was discussed in Section IV, the DataCutter filter-stream programming model provides the abstraction of filter group instances and transparent copies, which can be separately or collectively employed to improve application performance. In the experiments, we varied the number of group instances and the number of copies per



(a)

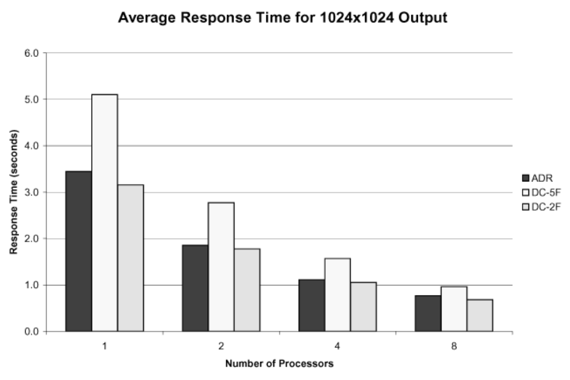


(b)

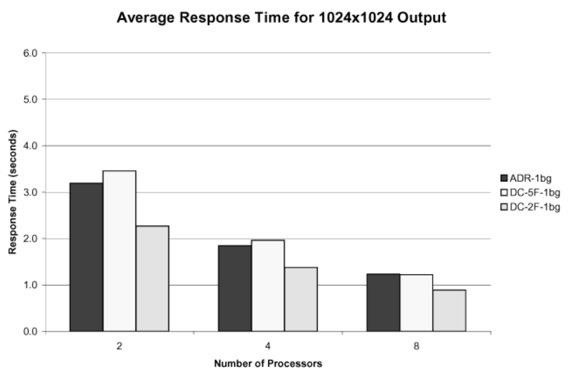


(c)

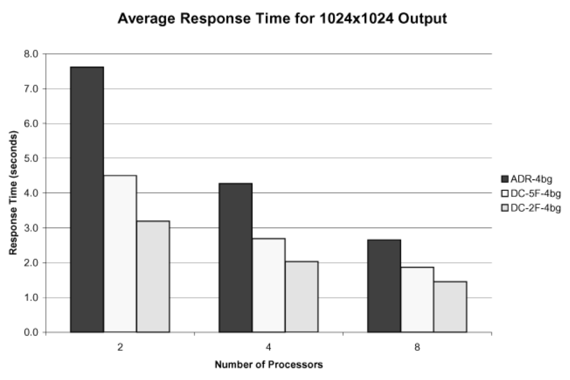
Fig. 11. Average response time of the VM servers for 512 × 512 output image.



(a)

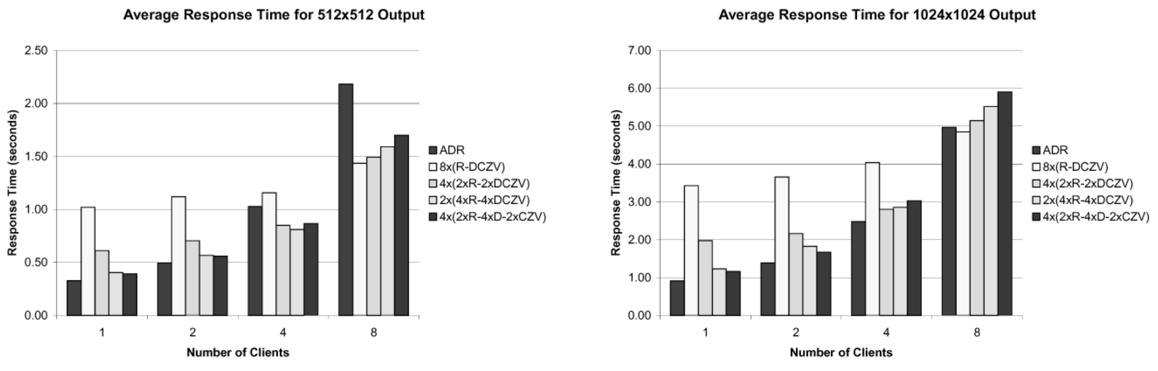


(b)



(c)

Fig. 12. Average response time of the VM servers for 1024 × 1024 output image.



(a) (b)

Fig. 13. Average response time of different versions of the VM servers on the 8-processor SMP.

filter in each group instance to create different server configurations. In the figures, $a \times (b \times F1 + c \times F2 + d \times F3)$ denotes that there are a instances of the filter group that consist of filters $F1, F2$, and $F3$ and that b transparent copies of $F1, c$ transparent copies of $F2$, and d transparent copies of $F3$ are executed for each group instance. Queries submitted by the clients are assigned to the group instances in round-robin fashion. Multiple queries can be executed concurrently if there is more than one group instance, but each group instance evaluates one query at a time. As is seen from the figure, for a small number of clients (e.g., 1 or 2) the ADR server performs better than the DataCutter server versions. When there are a few clients, it is more beneficial to execute each query in parallel using the maximum number of processors available. Similarly, the DataCutter server configurations with fewer instances, but more transparent copies per instance, achieve better performance for small numbers of clients. When the number of clients increases, the DataCutter implementations of the VM server with multiple group instances perform better. For a large number of clients, inter-query parallelism can be exploited to improve the performance of the data server. As is seen from Fig. 13(a), especially when the queries are small (i.e., the query window is small), there is more interquery parallelism available for improving application performance, as the parallel execution of a single query may incur load imbalance. When bigger queries are executed, it is likely that good load balance will be achieved for each query. Hence, the ADR server scales better for bigger queries, as is seen from Fig. 13(b). Nevertheless, our results show that DataCutter provides sufficient flexibility so that the data server configuration can be modified to accommodate various data access patterns.

V. Data Caching in the Client

In a client-server environment, the data server often needs to interact with many clients simultaneously. This can cause high demand on the server and network resources. For interactive applications the system as a whole should achieve acceptably small response times. Response time is measured as the amount of time between the initiation of a request and when the last piece of data is delivered. If a response takes too long, the application may become unusable. The base Virtual Microscope implementation attempts to achieve low response time by using a scalable parallel server with a disk farm so that data access and processing required by a client request can be completed quicky. However, for a given server configuration, as more clients are added, the server has to multiplex between more client requests. As a result, the response time observed by an individual clients rises. Moreover, if the connection between a client and the server spans a wide-area network, the demands on the network can be extremely high, and the latency incurred between clients and a server may be very high. In order to improve the overall system performance, the following issues should be addressed:

Reduction of wide-area network usage. Clients connected to a server over a wide-area network make use of resources that are shared by other applications, namely the long-haul links and intermediate nodes in the network. Transferring high volumes of data through such shared resources is expensive, thus the latency between a client and the server may be very high. Although the use of a parallel data server makes it possible to efficiently store and process very large images at the server, the output (a 2-D image) of a typical query may still be large. For instance, the size of the output is about 1 MB for a query window at 640 $\times$ 480 pixels resolution. Therefore it may not be possible to achieve interactive viewing for clients connecting to the server over a slow network connection. Using image compression techniques, the size of the output image can be reduced by perhaps a factor of 10, but even then a single view may require 20 to 30 seconds to transmit over a standard modem connection. Therefore compression cannot be the only mechanism used to reduce wide-area data volume for Virtual Microscope style applications.

Reduction in the server workload. Reduced workload at the server is an important way to improve overall system scalability. With less applied workload, the system should see reduced utilization, hence better performance as more clients are added.

One possible approach that can be taken to address these issues is to cache data near a client. In earlier work [5], we examined the performance impact of a diskless proxy, where data blocks were cached only in processor memory. A proxy behaves as an intermediate server between a set of co-located clients and a remote server. It appears to the remote server as a client, and to a client as a server.

Given that there is sufficient common interest among multiple clients, several benefits can be realized with a proxy in place. First, the response time seen by each client can be reduced. With efficient data caching, most client requests can be served by the proxy across the local-area network, instead of the data server across the wide-area network. Second, the amount of redundant data sent across the wide-area network can be reduced. Instead of multiple clients requesting the same or overlapping regions of a dataset from the server, the proxy can request the data only once. Third, server scalability is improved by reducing its utilization.

Our experimental results showed that using a proxy does improve overall system performance. However, caching at the proxy provides a benefit only when two requirements are met. First, the clients need to be local to the proxy. This reduces the long latency seen in contacting the remote server to resolve proxy cache misses. Second, some commonality of interest among the set of clients must exit. This reduces the working set size in the proxy, which helps avoid cache overflow. With both conditions satisfied, the maximum benefit occurs when all but the first request for a block of data is in the proxy cache. Moreover, since all client requests go through the proxy, response time as seen by the client for requests that miss the cache is higher than when there is no proxy.

Typical users of the Virtual Microscope (e.g., pathology consultants, medical students) would also like to access the image data via the Internet from their home, usually over a slow connection. In that case, clients can be at geographically distant locations. Moreover, compared to a collaborative environment, it is unlikely that there will be many overlapping regions of interest amount the clients. As a result, there will be little benefit from the use of a proxy. Nevertheless, a client can still benefit from data caching, if data can be cached at the client. For this purpose, we have designed the VM client to maintain and use a two-level cache; the client memory is a first-level cache, and the local disk on the client machine is a second-level cache. The caching mechanism implemented in the client works as follows. Images are viewed as partitioned into tiles at the client, where a tile is a fixed sized rectangular portion of the image.[1] . The tiles are used as the units of caching for portions of the image. When the user selects a region to view, the client determines the set of tiles that intersect with the selected region, and tiles that are in the cache are displayed directly. A least recently used (LRU) policy has been adopted for both levels of the cache. When a tile is needed to display a selected region of the image, the memory cache is first searched. If the tile is not found in the memory cache, the disk cache is searched and if the tile is found it is both inserted in the memory cache and used for display. Only tiles that are not in either cache are requested from the server, and before display are inserted into both the memory and disk cache.

A. Multiresolution Image Caching

The VM client caches an image at the resolution the image was retrieved from the server. Hence, when a user selects a region of interest, the client cache may contain multiple tiles at different magnification levels that intersect the region of interest. The VM client will first try to construct the requested image using the tiles in the cache. If the cache already contains all the corresponding tiles at the requested resolution, the output image is constructed from those tiles and displayed. However, part of the image may not be available at the requested resolution. In that case, the VM client first tries to construct the missing parts of the output image using tiles cached at higher resolutions (using the same subsampling algorithm as in the data server to construct the desired lower resolution image). If the entire image cannot be constructed with either tiles at the requested resolution or tiles at higher resolutions, the missing tiles are temporarily displayed by blowing up (replicating pixels from) the lower resolution tiles or, as a last resort, the thumbnail image. The parts of the image that have not been displayed at the desired resolution are requested from the data server. Fig. 14 shows these execution steps. In Fig. 14(a), the user-selected region is displayed from tiles that are available in the client cache. The lower left part of the requested image is displayed from tiles cached at the requested resolution or from tiles cached at higher resolution. The upper left part of the image is displayed from cached lower resolution tiles, and the right part of the image is displayed by blowing up the thumbnail image. The VM client requests the parts of the image that have not been displayed at the requested resolution from the server. In Fig. 14(b) the upper left part of the image is being replaced with the results from queries sent to the VM server. The client continues to fill in the image with tiles requested from the server for the right side of the image, as displayed in Fig. 14(c).

In the current implementation, the VM client sends queries to the VM server only at the user-specified resolution or at the next higher resolution. For example, suppose a user selected a region at 100$\times$ in a slide that was scanned and stored at 400$\times$. Fig. 15 displays an illustration of the slide tiling at three resolutions (100$\times$, 200$\times$, and 400$\times$). Although the user selection may intersect more than one tile at the requested resolution, for the sake of simplicity in the presentation, suppose the user selected a region that intersects only one tile, and let the shaded tile at 100$\times$ be that tile. If the tile is already in the cache, the requested region can be directly drawn using that tile. However, if it is not in the cache, the VM client searches for tiles at the next higher resolution that can be used to construct the tile at the requested resolution. The four tiles at 200$\times$ that can be used tro construct the requested tile at 100$\times$ are also shaded in Fig. 15. If the client cache contains all four of those tiles, the VM client can immediately draw the user-requested image via subsampling of the 200$\times$ tiles. If any of those tiles is not in cache, the client recursively searches at the next higher resolution (400$\times$) to construct the missing 200$\times$ tile. If the tile at 100$\times$ still cannot be fully constructed, the client must request tiles from the data server. The tile at 100$\times$ will be requested from the server if two or more tiles are missing at 200$\times$. However, if only one tile is missing at 200$\times$, the client will request the tile at 200$\times$ to attempt to reduce the workload at the data server, because a 200$\times$ tile will require fewer data chunks to be retrieved and proceed at the server than will a 100$\times$ tile.

For efficient use of memory and disk space resources on the client machine, images tiles are stored in JPEG format. This introduces compression and decompression overload when a tile is used. However, with JPEG compression we have been able to

[1]The best choice of image tile size in the client takes into account the tile size used in the data server. A client can request that information when it initially connects to the server.

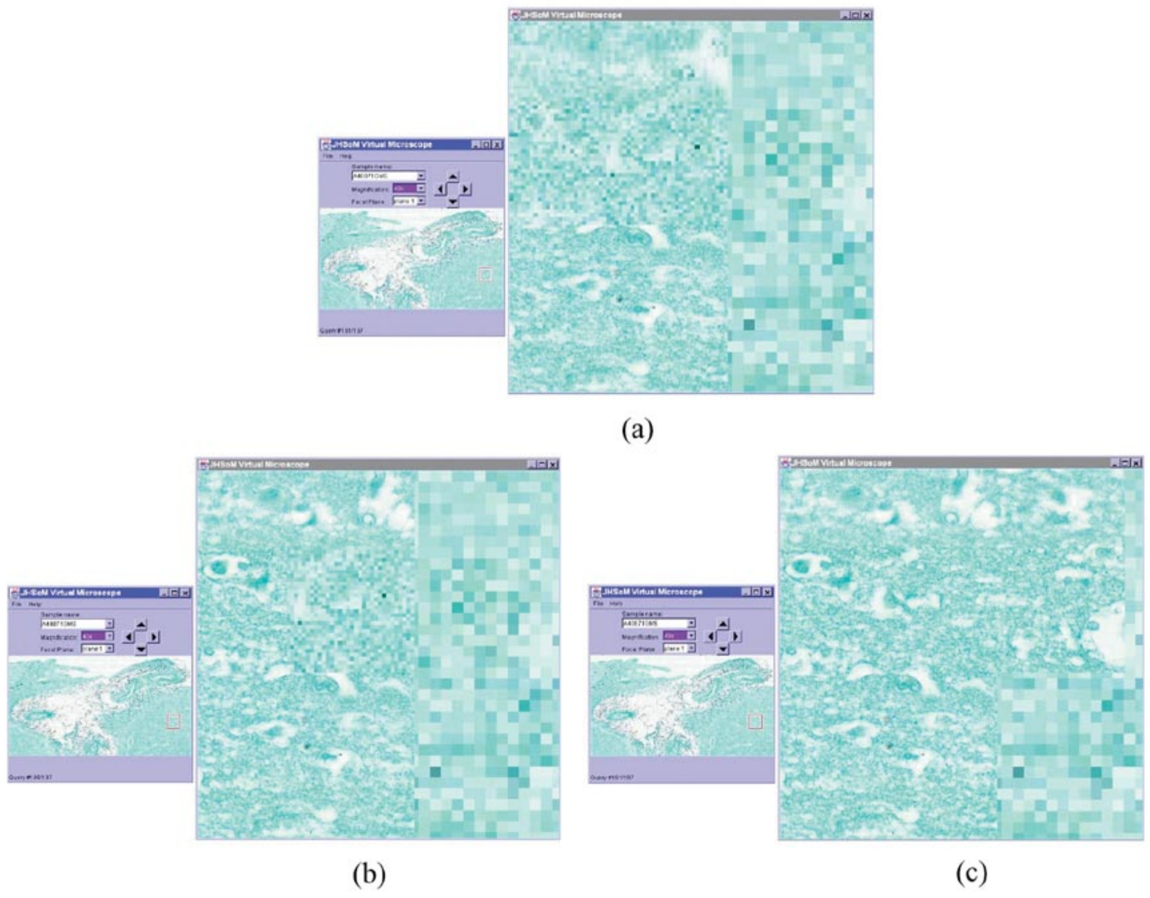


Fig. 14. Virtual Microscope client query execution steps using the multiresolution image cache.

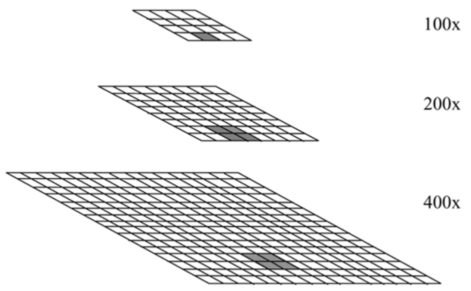


Fig. 15. An illustration of image tiling at multiple resolutions.

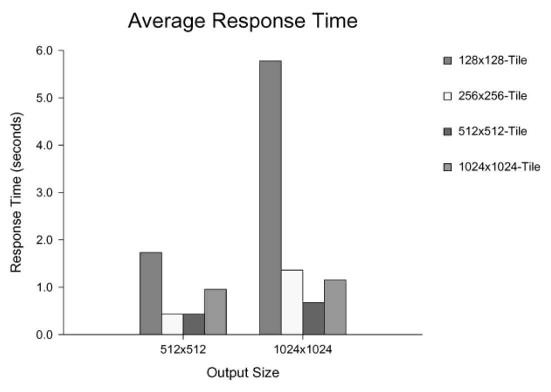


Fig. 16. Overall average response times of caching client for varying cache tile sizes.

compress the slide images by up to a factor of 15 from the original size without a noticeable loss in image quality. This helps to reduce the I/O time for each tile and more than compensates for compression/decompression overhead. Compression also allows for caching many more tiles for a given cache size, further improving overall system performance.

B. Experimental Results

For the experimental evaluation of caching client performance, we employ an ADR version of the VM data server, with the client generated workload the same as the described in Section III-C. The data server runs on the five processor PC cluster described in Section III-C. Average response times for the VM clients are displayed in Figs. 16–18. For the caching client, the response time also include caching overheads, such as cache lookups and inserting tiles into the cache. To insert a tile into the cache, server tiles received from the VM data server are decompressed and stitched together as required (no stitching is necessary if the client tiles are chosen to be the same as the server tiles), then the constructed tile is compressed and inserted into the client cache.

Fig. 16 displays the average response times of the caching client using different cache tile sizes, for queries that request 512×512 or 1024×1024 output images. Each bar in the figure shows the average response times for 500 queries using four different cache tile sizes. As is seen in the figure, for queries

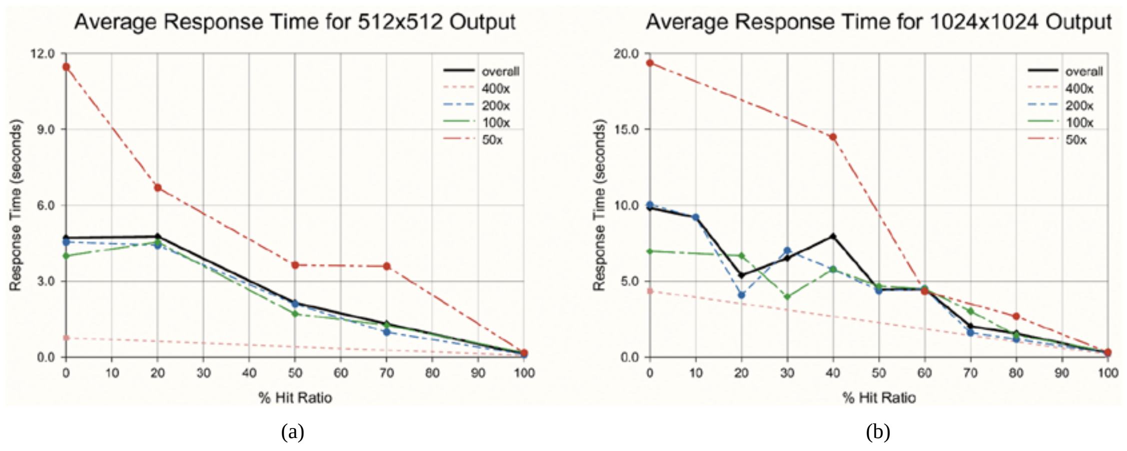


Fig. 17. Average response times of caching client for 1000 queries, generating (a) 512 × 512 output image, and (b) 1024 × 1024 output image. In this experiment, the cache tiles are 512 × 512.

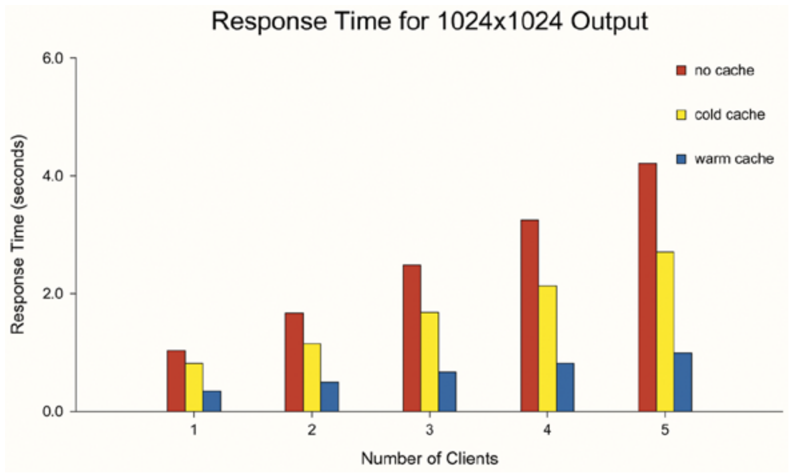


Fig. 18. The average response times of the noncaching and caching clients. The VM data server is run on five processors.

that request a 512 × 512 image, tile sizes of either 256 × 256 or 512 × 512 result in the best performance, with an average response time of about 0.4 seconds. For queries with an output image size of 1024 × 1024, a tile size of 512 × 512 produces the best performance, with about a 0.7-s average response time. Using a cache tile size that is too small causes many small requests to be sent to the VM server, which decreases server performance. Using a cache tile size that is too large causes the data server to process too much extra image data, and also increases client caching overhead because of JPEG compression and decompression of the tiles.

Fig. 17 displays the average response times of the caching client with respect to cache-hit ratio (i.e., the number of tiles obtained from cache divided by the total number of tiles requested), for output image sizes of 512 × 512 and 1024 × 1024. As we described in the previous section, the caching client first determines which tiles intersect with the user query and those tiles are searched for in the cache. We plotted the queries according to the percentage of tiles that intersect with the query that are available in cache versus average response time, with the results shown in Fig. 17. Each line plotted in the figure displays the average response times for queries at varying resolutions. The thick solid black line shows the *overall* average response times for each hit-ratio. As expected, the average response time of the VM client decreases drastically with increasing cache hit ratio. For example, for queries producing a 1024 × 1024 image at 50×, if none of the required tiles are in the cache the average response time can be as high as 19 seconds. However, if all tiles are in the cache, the average response time is only 0.3 seconds.

A comparison of the response times of the noncaching and caching clients is displayed in Fig. 18. For the caching client, two response times are displayed for each experiment. The first one shows the response time of the caching client when the client starts with an empty cache, labeled *cold cache*. The second one shows the response time of the caching client when the client has been run a second time, that is when it starts with a nonempty cache, labeled *warm cache*. For a single client, using the caching client starting with a cold cache reduces response time about 20% on average. The use of the caching client also improves overall VM system performance by reducing the load at the VM data server. For example, with five caching clients the average response time seen by each client is about 35% less than the average response time seen by five noncaching clients. Starting from a nonempty cache speeds up the response time of the caching client by more than 50%, leading to approximately 75% faster response time than the noncaching client, on average.

VI. Conclusion

In this paper we have discussed the design and implementation of a client/server database system that provides a realistic emulation of a high power light microscope. The system also provides capabilities that can never be achieved with a physical microscope, such as simultaneous viewing and manipulation of the same slide by multiple end users. To solve the main problem in providing a system that performs adequately, namely storing and processing *very* large quantities of slide image data, we have presented the design and implementation of two versions of the Virtual Microscope server software that target 1) tightly coupled parallel computers with a disk farm and 2) distributed computing environments providing access to archival storage systems. Both servers employ more general software frameworks, the Active Data Repository and DataCutter, appropriately customized to provide the required Virtual Microscope functionality. The use of such frameworks allows the server systems to take advantage of all the performance optimizations that have been engineered into the frameworks for executing a large class of data intensive applications on the targeted computational platforms. In addition, we have described the optimizations required in the client software to provide rapid response times for users, in particular caching image data in both memory and local disk on the client machine. The overall performance results show that the resulting Virtual Microscope system can provide scalable server performance and good client response times. Such results show that it is becoming feasible to deploy such a system within a clinical setting, for example allowing a pathologist to access a slide sample at any time from an inexpensive PC, without requiring physical access to a slide or a microscope.

References

[1] M. Aeschlimann, P. Dinda, J. Lopez, B. Lowekamp, L. Kallivokas, and D. O'Hallaron, "Preliminary report on the design of a framework for distributed visualization," in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, Las Vegas, NV, June 1999, pp. 1833–1839.

[2] A. Afework, M. D. Beynon, F. Bustamante, A. Demarzo, R. Ferreira, R. Miller, M. Silberman, J. Saltz, A. Sussman, and H. Tsang, "Digital dynamic telepathology—The virtual microscope," in *Proc. 1998 AMIA Annual Fall Symposium*, Nov. 1998.

[3] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The R^*-tree: An efficient and robust access method for points and rectangles," in *Proc. 1990 ACM-SIGMOD Conference*, May 1990, pp. 322–331.

[4] M. Beynon, T. Kurc, A. Sussman, and J. Saltz, "Design of a framework for data-intensive wide-area applications," in *Proc. 9th Heterogeneous Computing Workshop (HCW2000)*, May 2000, pp. 116–130.

[5] M. Beynon, A. Sussman, and J. Saltz, "Performance impact of proxies in data intensive client-server applications," in *Proc. 1999 International Conference on Supercomputing*, June 1999.

[6] M. D. Beynon, R. Ferreira, T. Kurc, A. Sussman, and J. Saltz, "Datacutter: Middleware for filtering very large scientific datasets on archival storage systems," in *Proc. Eighth Goddard Conference on Mass Storage Systems and Technologies/17th IEEE Symposium on Mass Storage Systems*, NASA/CP 2000-209 888, Mar. 2000, pp. 119–133.

[7] M. D. Beynon, T. Kurc, U. Catalyurek, C. Chang, A. Sussman, and J. Saltz, "Distributed processing of very large datasets with Data-cutter," *Parallel Comput.*, vol. 27, no. 11, pp. 1457–1478, Oct. 2001.

[8] M. D. Beynon, T. Kurc, A. Sussman, and J. Saltz, "Optimizing execution of component-based applications using group instances," in *Proc. IEEE Int. Symposium on Cluster Computing and the Grid (CCGrid2001)*, Brisbane, Australia, May 2001.

[9] M. D. Beynon, A. Sussman, U. Catalyurek, T. Kurc, and J. Saltz, "Performance optimization for data intensive grid applications," in *Proc. Third Annual International Workshop on Active Middleware Services (AMS2001)*, Aug. 2001.

[10] JJ2000 [Online]. Available: http://jj2000.epfl.ch

[11] C. Chang, R. Ferreira, A. Sussman, and J. Saltz, "Infrastructure for building parallel database systems for multi-dimensional data," in *Proc. Second Merged IPPS/SPDP Symposiums*, Apr. 1999.

[12] A. Chervenak, I. Foster, C. Kesselman, C. Salisbury, and S. Tuecke. (1999) The Data Grid: Toward an Architecture for the Distributed Management and Analysis of Large Scientific Datasets. [Online]. Available: http://www.globus.org/

[13] D. Comaniciu, W. Chen, P. Meer, and D. J. Foran, "Multiuser workspaces for remote microscopy in telepathology," in *Proc. IEEE Computer-Based Medical Systems*, vol. 2, 1999, pp. 150–155.

[14] G. Edjlali, A. Sussman, and J. Saltz, "Interoperability of data parallel runtime libraries," in *Proc. Eleventh International Parallel Processing Symposium*, Apr. 1997.

[15] C. Faloutsos and P. Bhagwat, "Declustering using fractals," in *Proc. The 2nd International Conference on Parallel and Distributed Information Systems*, San Diego, CA, Jan. 1993, pp. 18–25.

[16] R. Ferreira, T. Kurc, M. Beynon, C. Chang, A. Sussman, and J. Saltz, "Object-relational queries into multi-dimensional databases with the Active Data Repository," *Parallel Processing Lett.*, vol. 9, no. 2, pp. 173–195, 1999.

[17] R. Ferreira, B. Moon, J. Humphries, A. Sussman, J. Saltz, R. Miller, and A. Demarzo, "The virtual microscope," in *Proc. 1997 AMIA Annual Fall Symposium*, Oct. 1997, pp. 449–453. Also available as University of Maryland Technical Report CS-TR-3777 and UMIACS-TR-97-35.

[18] D. J. Foran, D. Comaniciu, P. Meer, and L. A. Goodell, "Computer-assisted discrimination among malignant lymphomas and leukemia using immunophenotyping, intelligent image repositories, and telemicroscopy," *IEEE Trans. Inform. Technol. Biomed.*, vol. 4, pp. 265–273, Dec. 2000.

[19] I. Foster and C. Kesselman, *The GRID: Blueprint for a New Computing Infrastructure*: Morgan-Kaufmann, 1999.

[20] Global Grid Forum [Online]. Available: http://www.gridforum.org

[21] A. Guttman and R. Trees, "A dynamic index structure for spatial searching," in *Proc. 1984 ACM-SIGMOD Conference*, June 1984, pp. 47–57.

[22] The Independent JPEG Group's JPEG Software (1998, Mar.). [Online]. Available: http://www.ijg.org

[23] JasPer vl. 6 (2000). [Online]. Available: http://www.ece.ubc.ca/mdadams/jasper/

[24] Interscope Technologies (2001). [Online]. Available: http://www.interscopetech.com

[25] C. Isert and K. Schwan, "ACDS: Adapting computational data streams for high performance," in *Proc. 14th International Parallel & Distributed Processing Symposium (1PDPS 2000)*. Cancun, Mexico, May 2000, pp. 641–646.

[26] ISO/IEC FCD 15 444-1: Information Technology—JPEG 2000 Image Coding System: Core Coding System [WG 1 N 1646] (2000, Mar.). [Online]. Available: http://www.jpeg.org/FCD15444-1.htm

[27] W. Johnston and B. Tierney, "A distributed parallel storage architecture and its potential application within EOSDIS," in *Proc. The Fourth NASA Goddard Conference on Mass Storage Systems and Technologies*, 1995.

[28] B. Moon, H. Jagadish, C. Faloustsos, and J. Saltz, "Analysis of the clustering properties of the Hilbert space-filling curve," *IEEE Trans. Knowledge Data Eng.*, vol. 13, no. 1, pp. 124–141, Feb. 2001.

[29] B. Moon and J. H. Saltz, "Scalability analysis of declustering methods for multidimensional range queries," *IEEE Trans. Knwoledge Data Eng.*, vol. 10, no. 2, pp. 310–327, Mar./Apr. 1998.

[30] S. Olsson and C. Busch, "A national telepathology trial in Sweden: Feasibility and assessment," *Arch. Anat. Cytol. Pathol.*, vol. 43, pp. 234–241, 1995.

[31] B. Plale and K. Schwan, "dQUOB: Managing large data flows using dynamic embedded queries," *Proc. IEEE International High Performance Distributed Computing (HPDC)*, Aug. 2000.

[32] M. Rodríguez-Martínez and N. Roussopoulos, "MOCHA: A self-extensible database middleware system for distributed data sources," in *Proc. 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD00)*, vol. 29, May 2000, pp. 213–224. ACM SIGMOD Record.

[33] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra, *MPI: The Complete Reference*, ser. Scientific and Engineering Computation Series: MIT Press, 1996.

[34] SRB: The Storage Resource Broker [Online]. Available: http://www.npaci.edu/DICE/SRB/index.html

[35] M. Teller and P. Rutherford, “Petabyte file systems based on tertiary storage,” in *Proc. The Sixth NASA Goddard Space Flight Center Conference on Mass Storage Systems and Technologies, Fifteenth IEEE Symposium on Mass Storage Systems*, 1998.
[36] R. Thakur, A. Choudhary, R. Bordaweakr, S. More, and S. Kuditipudi, “Passion: Optimized I/O for parallel applications,” *IEEE Computer*, vol. 29, no. 6, pp. 70–78, June 1996.
[37] J. Z. Wang, J. Nguyen, K.-K. Lo, C. Law, and D. Regula, “Multiresolution browsing of pathology images using wavelets,” in *Proc. 1999 AMIA Annual Fall Symposium*, Nov. 1999, pp. 340–344.
[38] M. Weinstein and J. I. Epstein, “Telepathology diagnosis of prostate needle biopsies,” *Human Pathol.*, vol. 28, no. 1, pp. 22–29, Jan. 1997.
[39] R. S. Weinstein, A. Bhattacharyya, A. R. Graham, and J. R. Davis, “Telepathology: A ten-year progress report,” *Human Pathol.*, vol. 28, no. 1, pp. 1–7, Jan. 1997.
[40] A. W. Wetzel, P. L. Andrews, M. J. Becich, and J. Gilbertson, “Computational aspects of pathology image classification and retrieval,” *J. Supercomput.*, vol. 11, pp. 279–293, 1997.

Ümit Çatalyurek received the Ph.D., M.S., and B.S. degrees in computer engineering and information science from Bilkent University, Turkey, in 2000, 1994, and 1992, respectively.

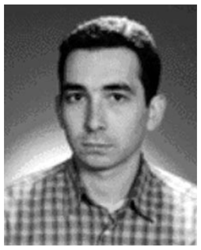
He is an Assistant Professor in the Department of Biomedical Informatics at The Ohio State University, Columbus. His research interests include graph and hypergraph partitioning algorithms, grid computing, and run-time systems and algorithms for high-performance and data-intensive computing.

Michael D. Beynon received the Ph.D. and M.S. degrees in computer science from the University of Maryland, College Park, in 2001, and 1998, respectively, and the B.S. degree in computer science from University of New York at Albany in 1994.

He is a Staff Scientist in the Advanced Networks and Applications Group at MIT Lincoln Laboratory. His research interests include runtime systems for data intensive applications, high-performance parallel and distributed systems; computer vision algorithms, and video surveillance systems.

Chialin Chang received the Ph.D. degree in computer science from the University of Maryland, College Park, in 2001, the M.S. degree in computer science from the University of California at Los Angeles in 1991, and the B.S. degree in computer science and information engineering from the National Taiwan University in 1987.

He works at L Labs Inc., in Taiwan. His research interests include algorithms for data-intensive computing on parallel computers, I/O systems, and high-performance databases.



Tahsin Kurc received the Ph.D. degree in computer science from Bilkent University, Turkey, in 1997 and the B.S. degree in electrical and electronics engineering from Middle East Technical University, Turkey, in 1989.

He is an Assistant Professor in the Department of Biomedical Informatics at the Ohio State University, Columbus. His research interests include runtime systems for data-intensive computing in parallel and distributed environments, and scientific visualization on parallel computers.

Alan Sussman received the Ph.D. degree in computer science from Carnegie Mellon University, Pittsburgh, PA, in 1991 and the B.S.E. degree in electrical engineering and computer science from Princeton University, Princeton, NJ, in 1982.

He is an Assistant Professor in the Computer Science Department at the University of Maryland, College Park. His research interests include compilers and run-time systems for parallel computers, high-performance database and I/O systems, and medical informatics.

Joel Saltz received the B.S. degree in mathematics and physics from University of Michigan, Ann Arbor, in 1978. He received the M.D. and Ph.D. degrees in computer science from Duke University, Durham, NC, in 1985 and 1986, respectively.

He is Professor and Chair of the Department of Biomedical Informatics, Professor in the Department of Computer and Information Systems and a Senior Fellow of the Ohio Supercomputer Center. Prior to coming to Ohio State, he was Professor of Pathology and Informatics in the Department of Pathology at Johns Hopkins Medical School and Professor in the Department of Computer Science at the University of Maryland, College Park. His research interests are in the development of systems software, databases and compilers for the management, processing, and exploration of very large datasets.