Optimizing Execution of Component-based Applications using Group Instances *

Michael D. Beynon†, Tahsin Kurc‡, Alan Sussman†, Joel Saltz†‡

† UMIACS and Dept. of Computer Science
University of Maryland
College Park, MD 20742

‡ Dept. of Pathology
Johns Hopkins Medical Institutions
Baltimore, MD 21287

{beynon,kurc,als,saltz}@cs.umd.edu

Abstract

Applications that query, analyze and manipulate very large data sets have become important consumers of resources. With the current trend toward collectively using heterogeneous collections of disparate machines (the Grid) for a single application, techniques used for tightly coupled, homogeneous machines are not sufficient. Recent research on programming models for developing applications in the Grid has proposed component-based models as a viable approach, in which an application is composed of multiple interacting computational objects. We have been developing a framework, called filter-stream programming, for building data-intensive applications in a distributed environment. In this model, the processing structure of an application is represented as a set of processing units, referred to as filters. In earlier work, we studied the effects of filter placement across heterogeneous host machines on the performance of the application. In this paper, we develop the problem of scheduling instances of a filter group running on the same set of hosts. A filter group is a set of filters collectively performing a computation for an application. In particular, we seek the answer to the following question: should a new instance be created, or an existing one reused? *We experimentally investigate the effects of instantiating multiple filter groups on performance under varying application characteristics.*

1. Introduction

For the past several years we have witnessed a shift in how computational resources are used. As wide-area networks connecting computational resources get faster, an increasing number of applications are making collective use of computational resources across a wide-area network. There is a large body of research on building such computational grids and providing support for enabling execution of applications in a Grid environment. One of the consequences of the ability to use collections of powerful computers is that scientific and engineering simulations are generating unprecedented amounts of experimental data. In addition, vast amounts of data are being gathered by advanced sensors attached to various instruments such as satellites and microscopes at geographically distributed institutions. The result is very large datasets distributed across a wide-area network. The possibilities become even more intriguing when we note that disks continue to become larger and cheaper, making it possible to configure a large disk-based storage system at relatively low cost. For example, a PC cluster with 800GB storage space can be built with 5 PCs, each with two 80GB EIDE disks, for about $8K using off-the-shelf components. Given this low price, many such disk collections can be set up at multiple widely separated locations, giving rise to *islands of data*, where cheap archival storage systems are used to hold large locally generated datasets.

Applications that query, analyze and manipulate very large data sets are relatively well understood when using tightly coupled parallel machines or homogeneous clusters. With the current trend toward collective use of heterogeneous collections of disparate machines (the Grid) for a single application, the same techniques that work well for good performance in a tightly coupled, homogeneous environment are not sufficient. The Grid provides a powerful environment, yet introduces many unique challenges for applications. First, computational and storage resources can be at distributed locations in a wide-area network. Second, the characteristics, capacity and power of resources, including storage, computation, and network, can vary widely. Third, the distributed resources can be shared by many applications. These characteristics have several implications for developing efficient applications. An application should be

*This research was supported by the National Science Foundation under Grants #ACI-9619020 (UC Subcontract #10152408) and #ACI-9982087, and by the Office of Naval Research under Grant #N66001-97-C-8534.

structured to accommodate the heterogeneous nature of the Grid. Moreover, the application should be optimized in its use of shared resources and be adaptive to changes in their availability. For instance, it may not be efficient or feasible to perform all processing at a data server when its load becomes high. In this case, the efficiency of the application depends on its ability to perform application processing on the data as it progresses from the data source(s) to the client, and on the ability to move all or part of its computations to other machines that are well suited for the computation.

Recent research on programming models for developing applications in the Grid has converged on the use of component-based models [**?**, **?**, **?**, **?**, **?**, **?**], in which an application is composed of multiple interacting computational objects. In the DataCutter project [**?**, **?**, **?**], we are developing a framework, called filter-stream programming, for developing data-intensive applications in a distributed environment. This model represents 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 unidirectional pipes that deliver data in fixed size buffers.

In earlier work [**?**], we investigated the effects of placement decisions on the overall performance of data-intensive applications. Our results show that careful decomposition of the application processing structure into filters and placement of filters can have a significant impact on performance. The choice of placement represents a degree of freedom in affecting application performance by placing filters with affinity to data sources near the sources, minimizing communication volume on slow links, co-locating filters with large communication volume, and placing computationally intensive filters on less loaded hosts.

In this work, we are concerned with the problem of scheduling instances of filter groups, which more generally applies to any distributed component system allowing distributed execution. A filter group is a set of filters collectively performing a computation for an application. In particular we try to answer the following question: *should a new instance be created, or an existing one reused?* When a new filter group instance is created, resources are allocated to be used for processing new work. In contrast, reuse of an existing filter group instance will share the previously allocated resources with this new work. The basic tradeoff is between the cost of allocating more resources vs. the queuing delays caused by sharing resources. This instance creation decision should be made carefully to avoid overloading shared resources such as memory and processors, while still achieving good performance for the application. Our overall objective is to develop methodologies to automate the instance creation and scheduling process. In this paper, as a first step toward this goal, we investigate changes in overall application performance when multiple filter group instances are instantiated. We present experimental results under varying application scenarios.

2. DataCutter

DataCutter is a middleware infrastructure designed to provide support for subsetting and user-defined processing of large multi-dimensional datasets across a wide-area network. It provides two core services, an indexing service and a filtering service, on top of which application-specific services can be implemented. The indexing service provides support for accessing subsets of datasets via multi-dimensional range queries. To ensure scalability to very large datasets, DataCutter uses a multi-level hierarchical index scheme. The filtering service provides support for executing application-specific processing as a set of components, called filters, in a wide-area Grid environment. For this paper, we will only consider the DataCutter filtering service.

2.1. Application Model

Applications are invoked by the user, and start out as a single process on some host machine. This process is called the *console process*. The console process can use the DataCutter filtering service to instantiate and execute collections of user-defined components, called *filters*, on other hosts. The style of interaction between the filters and the console process is referred to as the application model, and is defined along two axes.

Detached vs. Pass-Thru The interaction between the console process and any filters created is divided into two cases. Detached is where filter output data is not handled by the console process. For example, output data may be processed in some manner within the filters, and consumed outright, or written to a disk file. Sorting of out-of-core data is an example of this type, where output is written directly by a filter to a file. In contrast, Pass-Thru is where filter output is consumed by the console process. Pass-Thru resembles an elaborate remote procedure call (RPC) or remote method invocation (CORBA, Java RMI), where the result is collected in the same process that initiated the remote call.

Stand-Alone vs. Client-Server This describes the entity that initiated the work that drives the filters. Stand-Alone represents cases where the console process initiates the work. An example of this is a simulation code that decides on new processing and the required data based on a current set of data. In contrast, Client-Server is the case where one process sends work to the console process, which, acting like a front-end, passes the work to the filters for processing. The work description in the Client-Server case is commonly referred to as a query. Database and image servers are common examples.

2.2. Filters and Streams

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 sub-classing a filter base class. This provides a well-defined interface between the filter code and the filter service. The interface for filters consists of an initialization function, a processing function, and a finalization function.

```
class MyFilter : 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. A stream also denotes a supply of data to and from the storage media, or a flow of data between two separate filters, or between a filter and a client. There are two kinds of streams in DataCutter: file streams and pipe streams. A file stream is used to access files. A pipe stream is the means of unidirectional data flow between two filters, from upstream filter to downstream filter. Bi-directional data exchange can be achieved by creating two pipe streams in opposite directions between two filters.

All transfers to and from streams are through a provided 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 a 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, such as Nexus [**?**].

3. Application Execution

The process of manually restructuring an application using the filter-stream model is referred to as *decomposing* the application. In choosing the appropriate decomposition, we need to consider the complete dataflow path from data generation to ultimate consumption and the target machine configuration, which can be a distributed collection of heterogeneous machines. The main goal is to achieve efficient use of limited resources in a distributed and heterogeneous environment. Once the application is decomposed into a set of filters, the filters need to be placed on the set of host machines accessible to the application, and be instantiated to carry out application-specific processing.

3.1. Unit of Work

Filter operation progresses as a sequence of cycles, that each handle a single application-defined *unit-of-work*. An example of a unit-of-work would be a spatial query for an image processing application that describes a region within an image to retrieve and process. A work cycle starts when the filtering service calls the **init** function, which is where any required resources such as memory or disk scratch space are pre-allocated. Next the **process** function is called to continually read data arriving on the input streams in buffers from the sending filters. A special marker is sent after the last buffer to mark the end for the current unit-of-work. The **finalize** function is called after all processing is finished for the current unit-of-work, to allow release of allocated resources such as scratch space. When a work cycle is completed, these interface functions may be called again to process another unit-of-work (see Figure 1).

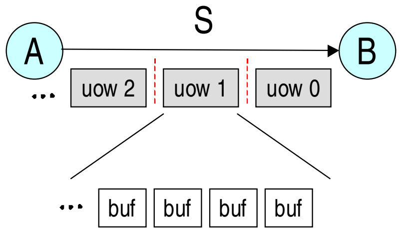


Figure 1. Data buffers and end-of-work markers on a stream.

3.2. Placement

Filters are the unit of placement, and each filter can potentially be executed on a different host. In addition, a filter's location may change at unit-of-work boundaries during the course of execution. Note this does not imply true migration of code and state, but rather placement can be recomputed and the original filter can be stopped on the original hosts and a new one created with a different placement, with a single buffer transfer supported per filter for application controlled restoration of state. This approach avoids many of the details and overhead involved in check-pointing and process migration [**?**]. Filters need to be structured appropriately to handle such events. For cases when a filter has some affinity to a particular host, the filter can be *pinned* to a particular host, so that the filter will always be placed on that host.

The basic idea behind the use of the filter-stream programming model is to somewhat 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. (1) 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 hosts to be controlled by the filtering service. For example, two filters with a large communication volume between should not be placed on opposite ends of a slow wide-area network connection. (2) Filters are expected to disclose and be granted memory and disk scratch space, instead of using unconstrained dynamic memory allocation. The granted scratch space is allocated on behalf of the filter by the runtime system when the filter is instantiated. For example, a filter can be run on a machine with enough memory to avoid paging, and two filters requesting large scratch space can be placed on separate machines.

3.3. Filter Groups

Once the placement is decided, filters must be instantiated to carry out application specific processing on the data. In many cases, resources are shared by multiple applications. In addition, an application or a set of applications may use the same filters or group of filters for executing multiple workloads. For instance, in a collaborative visualization environment, requests from multiple users can use the same set of filters, which collectively implement a visualization pipeline. Therefore, the number of instances of filters or filter groups to be created and the scheduling of filter instances are important decisions that affect overall performance.

The DataCutter filtering service provides the abstraction of a *filter group* to the console process. A filter group is a set of running filters that are logically related and are used together to perform a computation. For example, an application's console process may implement filters for performing various image processing operations: **anti-alias**, **subsample**, **reduce-colors**, and **clip**. One such logically related set of filters could be a processing chain that first **subsamples**, then **clips**. This description of two filters could then be executed as a filter group.

Multiple concurrent filter groups are supported by the DataCutter runtime system. This includes multiple filter groups that represent the *same* set of filters and streams, and/or filter groups comprised of different filters with different stream connectivity. Each filter within a filter group is executed in a separate thread of control (i.e. a POSIX thread) to allow for concurrent execution. The console process can then append work to any running instance it has created. Work is handled in FIFO order by an instance. There is no ordering between work appended to concurrent filter instances. The current implementation requires a running filter to be a member of exactly one filter group. Figure 2 provides a summary illustration of a filter group comprised of filters **A** and **B** with a single stream **S** for communication. This filter group is instantiated twice, each using a different placement. The filters for the first instance are all located on host1, and denoted by subscript 0, and the filters for the second instance are located on both hosts, denoted by subscript 1.

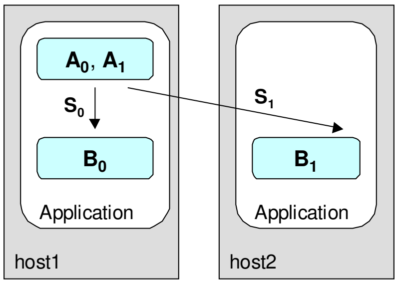


Figure 2. Two identical filter group instances running with different placement.

In deciding how many instances to create, the basic tradeoff is between host resources and latency. If a new work item is appended to an existing instance, this can potentially add queuing delay to the response time of the new work item due to FIFO processing of previous work items. Creating a new instance in this case will avoid the queuing delay, but does so by concurrently running another filter that can saturate shared resources such as processors, physical memory, disk, etc. Since each concurrent instance processes its assigned workload independent of other instances, an application must be able to handle out-of-order delivery of results from a batch of work requests.

4. Experimental Results

In this section, we present experimental results on a Linux PC cluster, containing two-processor SMP nodes, connected via switched Gigabit Ethernet. We employed a PC cluster primarily to achieve a controlled environment for the experiments and to isolate the effects of changes in the availability of resources that are out of our control, such as variance in the network bandwidth due to other users in a wide-area network.

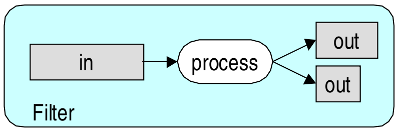


Figure 3. Application emulator filter model.

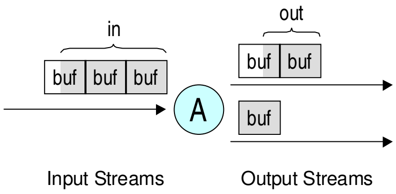


Figure 4. Use of fixed-size buffers.

We have implemented a filter emulator, that abstracts the processing and data handling we have seen through implementing various DataCutter applications. The advantage of using a filter emulator is we can easily adjust application characteristics to fully explore the large space of potential application filters. The emulator itself assumes a particular model of filter operation, as seen in Figure 3. An emulated filter operates in strict dataflow style: the filter (1) blocks to read sufficient input on *all* its input streams, (2) performs computation on the input, and (3) generates some amount of output data to write to *all* its output streams. All input and output operations are performed using fixed size buffers (Figure 4).

In the experiments, we used a simple application that implements three filters connected as a linear chain. **P** - the producer that reads blocks of data from a local disk, **F** - an inline filter that process the data in some manner reducing its size by half, and **C** - a consumer that finishes the processing of the data and discards it. These three filters form a filter group.

In this paper, filter placement is statically chosen for each execution; each filter in a filter group is placed on a separate cluster node to isolate its effects. We used the same set of nodes to instantiate new instances of the filter group so that when multiple instances were created more than one filter threads are executed on the corresponding nodes. All the filters were parameterized to create three application scenarios based on the ratio of computation to communication and I/O. (1) *I/O intensive* is where the ratio is less than 1, (2) *balanced* is where the ratio is about equal, and (3) *Cpu intensive* in which the ratio is larger than 1. The emulator parameter values selected are shown in Table 1. They were chosen to achieve the desired application behavior on the experimental platform, and to control running time of the experiments. For example, the **F** filter in the I/O intensive case will read $256kb$ from the input stream and write $128kb$ to the output stream (transform size), and take $0sec$ for processing the dataflow cycle (computation).

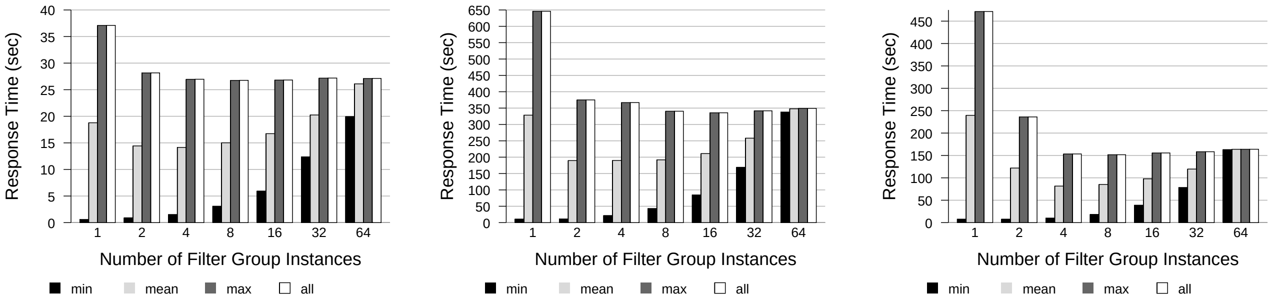
Response times were collected for each unit-of-work of a single run. In the figures, *min, max* and *avg* are the minimum, maximum and average response time across all units-of-work for the entire run, respectively. *all* is the complete time from the submission of the first unit-of-work to the completion of the last unit-of-work in an entire run. Note that *max* will be equivalent to *all* for experiments where all units-of-work are issued in a single batch. Which response time metric an application would attempt to minimize will depend on what is most important for that application. For example, a console process that is able to inject a set of work into running instances all at once, and cannot make progress until all responses are completed will need the lowest *all* metric. If in contrast, an application injects a single unit-of-work at a time and waits for a response before generating the next unit-of-work, then the *max* metric should be optimized. For most of the discussion that follows, we focus on the *all* metric.

Vary number of instances: Figure 5 shows the change in the response time with varying number of filter group instances for the three application scenarios. In these experiments, the number of instances was varied from 1 to 64. The workload was fixed at 64 units-of-work, all of which were submitted round-robin to the instances at the start of execution. As is seen from the figure, the performance of the application improves as more instances are created. This is expected since the total workload is partitioned across more filter groups. However, after an optimal point, the response time starts to increase because additional filter instances consume and eventually overload the available system resources such as cpu time.

Vary batch size: Figures 6 shows the response time and the optimal number of instances when 64 units-of-work are submitted in various batch sizes *per instance*. The optimal value is selected based on the lowest *all* response time value. A batch is appended to each filter instance after the previous batches have finished. While not shown in these figures, the general trend is that increasing the batch size for a *fixed* number of filter group instances improves performance. This is expected because doubling the batch size will result in one fewer latency penalty seen when waiting for all batch work to complete before issuing more work. As seen from the figure, the optimal number of instances changes as the number of batches is varied with no simple pattern. In addition, the optimal number of instances changes across the applications for a given set of batches.

Component	Parameters	I/O Intensive	Balanced	Cpu Intensive
Console	total bytes per unit-of-work	8 mb	1 mb	8 kb
	buffer size for all transfers	128 kb	32 kb	128 bytes
P	producer I/O: const+per kb	0 + 0/kb ms	10 + 1/kb ms	100 + 10/kb ms
	transform size: in, out	0, 256 kb	0, 64 kb	0, 256 bytes
	computation: const+per buf+per kb	0 + 0/buf + 0/kb ms	100 + 10/buf + 5/kb ms	100 + 20/buf + 10/kb ms
F	transform size: in, out	256 kb, 128 kb	64 kb, 32 kb	256 bytes, 128 bytes
	computation: const+per buf+per kb	0 + 0/buf + 0/kb ms	100 + 10/buf + 5/kb ms	100 + 20/buf + 10/kb ms
C	transform size: in, out	128 kb, 0	32 kb, 0	128 bytes, 0
	computation: const+per buf+per kb	0 + 0/buf + 0/kb ms	100 + 10/buf + 5/kb ms	100 + 20/buf + 10/kb ms

Table 1. Emulator parameters for console and filters for the three application scenarios.



(a) I/O intensive application

(b) Balanced application

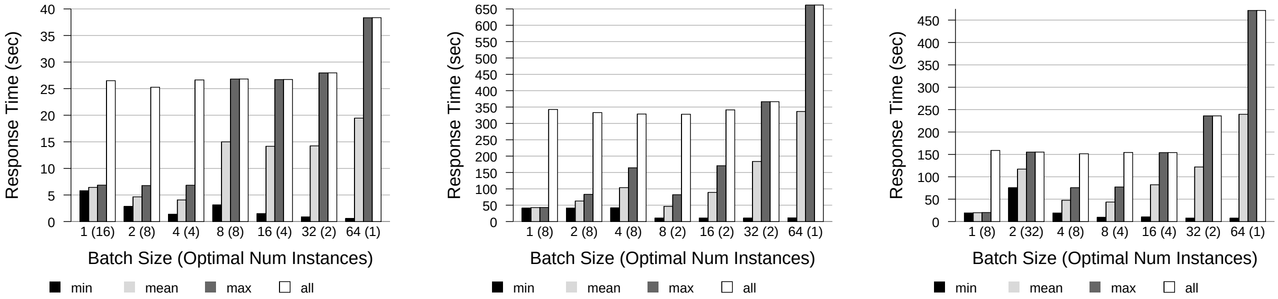
(c) Cpu intensive application

Figure 5. Change in response time for application scenarios, as number of filter group instances are varied. All 64 units-of-work are appended round-robin to filter group instances at the beginning.

Vary scratch memory use: Finally, we fixed the number of instances at 4 and set the total amount of work to 16, and examined the effect of varying the total amount of scratch memory allocated by filters on a host. As expected, the application takes a major performance penalty when the amount of scratch space in use exceeds the $256MB$ of physical memory available on the host (see Figure 7). As was seen in the other experiments, these results also vary across application types.

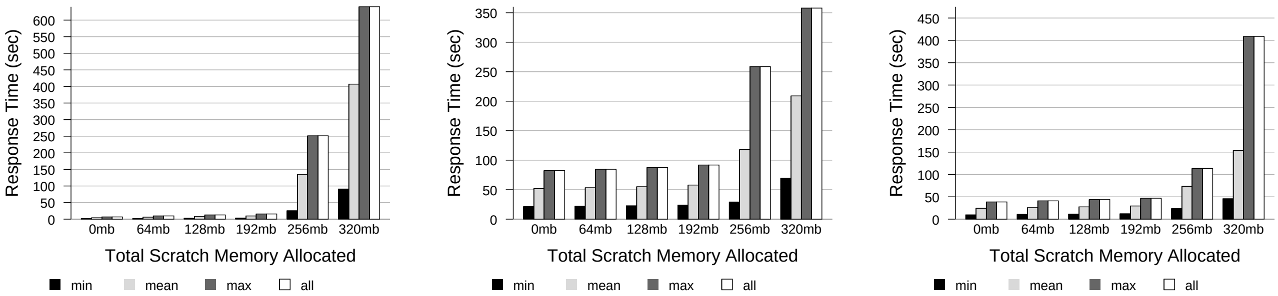
The premise of this paper is that optimizing performance depends on the ability to dynamically create additional filter group instances as needed. To achieve this goal, the costs involved in instance creation and assigning work to instances must be as low as possible. The time to create a new instance in all the experiments described, which requires validating the layout of filters and sending a command from the console to the remote host to execute a given filter, is on average $0.59sec$ on the PC cluster. Assigning work is simpler, requiring sending the work buffer to remote hosts containing filters for the instance, and is on average $0.0074sec$. These times do not include the time in the remote process for actually creating the filter object and thread, or for appending the work to a filter's work queue. The overall design of DataCutter has minimal barriers and other blocking operations, such that most operations are asynchronous, and block only when absolutely necessary. In particular, creating a new filter group instance is overlapped with the console process creating the next filter instance or with assigning work to existing filter instances. This approach is essential for any practical Grid framework that supports adaptivity.

The experimental platform was chosen to provide a degree of experimental repeatability, and to be able to isolate certain desired effects, while minimizing the impact of outside forces (such as background jobs, cross traffic, network contention, etc.). The results are relatively stable, and the lessons learned should still apply to a non-dedicated Grid environment. A problem that has not been addressed is that network characteristics can have a significant impact on performance. To address this, we have recently installed NistNet [**?**] on the PC cluster, which is a network emulator for wide-area networks. Appropriately configured, NistNet will allow us to emulate a wide-area network between any two host pairs in our cluster. This does not perturb either the sender or the receiver, because a special gateway node in the cluster does all the buffering and delaying, resulting in a controlled experimental platform that both provides repeata-



(a) I/O intensive application (b) Balanced application (c) Cpu intensive application

Figure 6. Change in the response time for application scenarios, as batch size is varied. For a given batch size, lowest "all" response time is used to choose the optimal number of filter group instances. The numbers in parentheses are the optimal number of instances for the corresponding batch size.



(a) I/O intensive application (b) Balanced application (c) Cpu intensive application

Figure 7. Change in the response time for application scenarios, as the total amount of scratch memory across filters is varied. For this experiment, the number of units-of-work is 16, and the number of filter groups instances is fixed at 4.

bility and also a controlled form of the real, chaotic Grid environment. For the final version of this paper, we plan to complete a version of our experiments using NistNet.

5. Related Work

All of the following work deals with data-intensive applications, and to varying degrees, how to execute them in a wide-area environment. To our knowledge, this paper is the first attempt to investigate the implications of using multiple instances of the same components on the performance of data-intensive applications in a Grid.

The ABACUS framework [**?**, **?**] addresses the automatic and dynamic placement of functions in data-intensive applications between clients and storage servers. This work is closely related to DataCutter in that application components are placed to improve performance, but ABACUS only support applications that are structured as a chain of function calls, and the only possibilities for placement are the client or server.

MOCHA [**?**] is a database middleware system designed to interconnect data sources distributed over a wide area network. MOCHA operates in the highly structured relational database world, and can automatically deploy implementations of new data types to hosts for execution of queries. The work shows how an optimizer customized to deal with "data-inflating" and "data-reducing" operators, can improve performance. MOCHA can leverage total knowledge about query selectivities stored in its catalog, whereas DataCutter deals with arbitrary application code with no such useful information.

ACDS [**?**] (Adapting Computational Data Streams) is a framework that addresses construction and adaptation of computational data streams in an application. Computational data streams model application processing as computational objects associated with data streams. A computational object performs filtering and data manipulation. Data streams characterize data flow from data servers or from running simulations to the clients of the application. The runtime system of ACDS performs migration, splitting and merging of streams to adapt for runtime variations such as data volume and availability of resources.

The dQUOB [**?**] (*dynamic QUery OBjects*) system consists of a runtime and compiler environment that allows an application to insert computational entities, called *quoblets* into data streams. The data streams targeted by the dQUOB system are those used in large-scale visualizations, in video streaming to a large number of end users, and in large volumes of business transactions.

Dv [**?**] is a framework for developing applications for distributed visualization of large scientific datasets on a computational grid. The Dv framework is based on the notion of *active frames*. An active frame is an application-level mobile object that contains application data, called *frame data*, and a *frame program* that processes the data. Active frames are executed by *active frame servers* running on the machines at the client and remote sites.

6. Conclusions and Future Work

In this paper, we address the problem of scheduling filter groups in an application. A filter group is a set of filters collectively performing computation on data. Our experimental results show that as the number of instances increase, the application performance improves because the workload is partitioned across parallel instances. However, the performance degrades after some optimal point because more instances will eventually overload system resources. In addition, we have demonstrated that application characteristics such as the ratio of computation to communication, and how the application injects work into the filter group instances, can affect the optimal choice. Therefore, we conclude that overall application performance is dependent on tailoring the number of filter group instances to application and machine characteristics, and that scheduling of filter instances is an important problem.

The DataCutter project is an ongoing attempt to provide an application component framework and runtime support for data-intensive applications, in a Grid environment. Our long term goal is to develop methodologies and various analytical cost models for selecting both a set of hosts and the right number of filter group instances for a given application. The cost models will be driven by important application and configuration parameters that were demonstrated in this paper and in previous work [**?**, **?**]. We plan to continue this initial study into the instance creation problem in a heterogeneous Grid environment and to develop emulators for other application classes, which will be also be used to develop and evaluate the cost models. Finally, we are extending DataCutter to allow instances of *individual* filters to be shared among separate filter group instances. This will add more dimensions to the filter group instance creation problem presented in this paper, because we can decide *per-filter* whether to create a new instance or multiplex an existing one.