ACDS: Adapting Computational Data Streams for High Performance

Carsten Isert Karsten Schwan

College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332
{isert,schwan}@cc.gatech.edu

GIT-CC-00-01

Abstract

Data-intensive, interactive applications are an important class of metacomputing (Grid) applications. They are characterized by large data flows between data providers and consumers, like scientific simulations and remote visualization clients of simulation output. Such data flows vary at runtime, due to changes in consumers' data needs, changes in the nature of the data being transmitted, or changes in the availability of computing resources used by flows.

The topic of this paper is the runtime adaptation of data streams, in response to changes in resource availability and/or in end user requirements, with the goal of continually providing to consumers data at the levels of quality they require. Our approach is one that associates computational objects with data streams. These objects offer services like data filtering and transformation. Runtime adaptation is achieved by adjusting objects' actions on streams, by splitting and merging objects, and by migrating them (and the streams on which they operate) across machines and network links. The resulting adaptive computational data streams maintain high performance by responding to changes in the needs of data consumers, as exemplified by variations in the resolution or rate at which they desire to receive data. Adaptive streams also react to changes in resource availability detected by online monitoring. The experimental demonstrations presented in this paper utilize computational data streams emanating from a global atmospheric simulation model and/or from stored model outputs, consumed by visualization clients that display this data. Experiments are performed on heterogeneous cluster machines and visualization clients connected by LAN or WAN networks.

1 Introduction

End users of high performance codes increasingly desire to interact with their complex applications as they run, perhaps simply to monitor their progress, or to perform tasks like program steering, or to collaborate with fellow researchers using these applications as computational tools. For instance, in our own past research, we have constructed a distributed scientific laboratory with 3D data visualizations of atmospheric constituents, like ozone, and with parallel computations that simulate ozone distribution and chemistries in the earth's atmosphere. While an experiment is being performed, scientists collaborating within this laboratory may jointly inspect certain outputs, may create alternative data views on shared data or create new data streams, and may steer the simulations themselves to affect the data being generated. Similarly, for metacomputing environments, Alliance researchers are now investigating and developing the Access Grid [S+99] framework and domain-specific 'portals' for accessing and using computations that are spread across heterogeneous, distributed machines.

The problem addressed by our research is the creation and management of the large-scale data streams existing in distributed high performance applications. The specific streams investigated in this paper are those emanating from data stores or from running simulations and consumed by visual displays that are employed by collaborating end users. Each such stream consists of a sequence of data events that flow from information providers to consumers, generated in response to requests from the consuming user interfaces and/or generated continuously by producers.

The event-based description of a data stream presented above provides a natural vehicle for associating computations with event generation, transport, and receipt, via event handlers located in producers, consumers, or in intermediate engines. The resulting *computational data streams* constitute the basis of our approach to online stream manipulation. Specifically, we adapt the behavior of the streams' events handlers in response to changes in end user capabilities or needs and/or in response to changes in resource availability. A simple adaptation example is one that uses a parameterized event handler to change the way in which stream data is sampled, perhaps to downsample it in order to fit it to the capabilities of a low end display engine. In addition to such client-driven adaptations, the computations performed on data streams may also be varied to cope with runtime changes in resource availability, as exemplified by a reduction in network bandwidth to some high end client, addressed by additional downsampling of the data being sent.

This paper demonstrates how computational data streams may be adapted in order to gain and maintain high performance in local and wide area systems. It also presents the ACDS framework for implementing adaptive computational data streams. ACDS supports runtime configuration actions that include (1) the migration of stream computations, (2) the specialization of these computations, in response to changes in end user needs and in resource availability, and (3) the splitting and merging of stream computations to increase and decrease concurrency as per a stream's runtime needs. Such actions are performed on computational data streams by control events. Control events are triggered (1) by changes in an end user's needs or behavior, and (2) by changes in resource availability captured by the ACDS system's distributed monitoring daemons.

The research contributions presented in this paper are: (1) the description of sample computational data streams used in distributed high performance applications; (2) the identification of opportunities for runtime stream adaptation; (3) the design and implementation of the ACDS system, providing adaptation support; and (4) the demonstration of the utility of ACDS for sample computational data streams, by improving the scalability of data streams and their utility for end users.

To summarize, ACDS supports the creation of adaptive computational data streams that transport precisely the data that is needed, when it is needed, at the levels of resolution and with transmission rates currently desired by end users.

The remainder of this paper first places ACDS in the context of related work in Section 2 and describes a sample scientific application and the computational data streams it uses (see Section 3). Section 4 identifies the opportunities for runtime stream manipulation existing for this application and its end users, followed by a description of stream computations and configuration actions that implement these manipulations. Section 5 describes the ACDS system abstractions and their realization. ACDS is evaluated in Section 6, where performance results demonstrate substantial advantages of adaptive vs. non-adaptive streams. In one case, stream performance is improved by 400% by migrating some of its components. In another case, stream performance is increased due to the use of additional parallelism for one of the stream's computationally intensive components. Conclusions and future work appear in Section 7.

2 Related Work

The ACDS system and its support for computational data streams rely on a high performance event infrastructure, called ECHo [Eis99], layered directly on transport-level communication protocols and capable of moving data at rates comparable to those of high performance programming platforms like MPI [MPI]. ECHo's event-based approach to defining and organizing data streams has been used in prior work on system monitoring and control [GES98, ES98]. In comparison to such work, ACDS employs ECHo for both transporting data and for controlling data transport. The computations associated with data events can apply both general or application-specific compression and filtering techniques to data, thereby giving ACDS the ability to improve system scalability, by reduction of processing and bandwidth needs and by provision of differential services to multiple end users. ECHo's event-based paradigm supports data streams with multiple producing or consuming subscribers; it can deal with dynamic subscriber arrivals and departures; and it supports runtime evolution for the types of data events a subscriber produces or consumes.

Past work on multimedia systems has already proven the efficacy of runtime stream control, by exploiting tradeoffs in stream throughput or fidelity vs. reliability in transmission, including the acceptance of significant loss rates in order to accommodate a larger number of users or to retain desired transmission and display rates despite variations in network and CPU availabilities [RKC99, WS99]. Beyond such performance results, much of the research funded by the Digital Library Initiative in the U.S. attempts to identify semantically relevant portions of media stream, based on which systems like ACDS could perform meaningful online filtering of data streams. Potential performance gains derived from such filtering are demonstrated in our past work on *Active User Interfaces* [IKS+99a].

The Interactivity Layer Infrastructure (ILI) [MS98] represents our previous approach to online data stream adaptation. This paper extends that work as follows. First, by using *Active User Interfaces* that emit control events describing current user behavior, ACDS can react to changes in end user needs as well as to changes in underlying system loads. Second, ACDS has been used in both LAN and WAN environments, thereby demonstrating its applicability to the Grid-based scientific computations and collaborations we aim to support. Third, the ACDS framework offers adaptation capabilities and support beyond that provided by ILI, including robust split and merge operations, decision algorithms, monitoring support, etc.

ACDS-controlled computational data streams may be used with arbitrary parallel applications, including those written with meta-computing systems like Globus [FK97], Legion [GW96], Schooner [HS94], or Active Harmony [HK98]. In these contexts, ACDS addresses only the runtime control of the computational data streams linking such Grid computations to end users. In comparison, the load-balancing and resource management mechanisms included with the grid computing frameworks themselves concern the runtime managent of the actual grid procedures, threads, or processes. It would be interesting to study how ACDS' data stream management interacts with load balancing performed for grid computations. Finally, both Cumulvs [GKP97] and ACDS customize task migration by use of application-specified knowledge. However, as with the load balancing performed in other metacomputing environments, Cumulvs does not know about entire data streams, nor does it support stream adaptations like component splitting or merging.

ACDS' implementations of the split and merge operations could take advantage of previous approaches to component checkpointing and migration, including those presented in [LTBL97, ZML99, Lud92]. However, the restrictions imposed by these approaches (e.g., homogeneity or hardware support) forced us to develop our own approach.

3 Sample Application

The sample application used in our research is a global atmospheric climate model, described in previous publications [KSS+96, JRZ+97]. The data streams of principal interest to this paper link the running model and/or data stored from previous model runs to visualizations employed by end users.

From our users' perspectives, useful views of this data display information about species concentration in grid form, where a grid point represents an area of approximately 5.6 x 5.6 degrees of latitude and longitude on the earth's surface. In order to provide this data view, however, the computational data stream producing it must first transform data from its model-resident or stored 'spectral' form to the grid-based form meaningful to end users. This transformation (termed 'Spec2Grid') may be performed on the receiving machine, on some intermediate node, or by the data producer. The resulting pipeline-structured computational data stream linking a single producer to two consumers is shown in Figure 1. This figure also shows an additional computation, termed 'Gridred', that filters the data being sent to the UI so that only those grid points currently requested by the end user are actually sent. The actual display processing is done in the elements termed AUI.

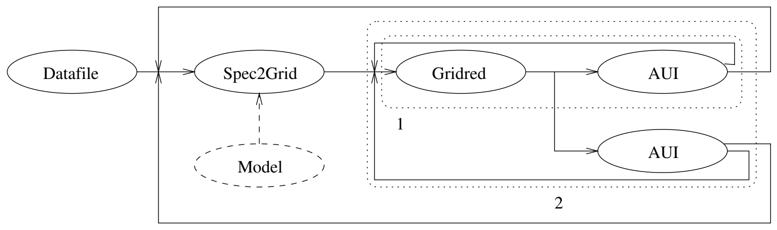

Figure 1: Sample Access Grid Application

The computations associated with the sample data stream shown in Figure 1 include data transformations required by the scientific end user, data compression needed for efficient storage or transport, data uncompression, data filtering or downsampling to capture current user needs and/or reduce transport costs, and data conversions with respect to display, the latter including view computations or additional graphical computations like isosurface determination [JRZ+97, ZS99]. Some of these computations are substantial, and so are their effects on the sizes of data events being transported. For instance, a typical spectral to grid transformation can be performed at the rate of 213 levels per second on a Sun Ultra 30, and the data expansion implied by this conversion increases the size of spectral data by a factor of 4.04 when producing grid data. This implies that it would be advantageous to postpone conversion until the data reaches the consumer, in order to preserve bandwidth. However, even high performance graphical rendering machines, like our OpenGL-based, feature-poor active UI running on an SGI Octane, can be overwhelmed by the processing and storage demands of a visualization that must render large data sets. This is one of the interesting problems to be addressed by the runtime methods for data stream configuration presented next.

4 ACDS: Concepts

4.1 Stream Adaptations

ACDS supports the construction and adaptation of computational data streams used in scientific applications. Since stream computation may themselves be computationally intensive, they can benefit from parallelization. This motivates ACDS' 'split' and 'merge' adaptations described below. Since the amounts of data being streamed may be large, data cannot be viewed in its entirety at all times. This implies the need for data filtering and the need to change filtering at runtime in accordance with current user behavior or needs; these needs motivate ACDS' support for the runtime adaptation of parameters in single or sets of stream components. Finally, ACDS supports the runtime migration of stream components, in order to deal with dynamic variations in the node and network loads of the underlying computational and access grids.

Parameter Changes. *Parameter changes* are actions that alter the behaviour of individual stream components. ACDS supports such changes with control events consumed by stream components and generated by user interfaces, by the ACDS monitoring and steering tool (MST), or by other stream components. The design and implementation of control-enabled stream components are described in more detail in Section 5.1 and in [IKS$^+$99a].

Task Migration. Dynamic load-balancing algorithms [Lud92, ZLP95, RD97, LMR91] may be classified by the distribution levels of their algorithms and by the ways in which they can affect application behaviour, ranging from local knowledge and local changes to global knowledge and global changes. ACDS enables task migration based on both local or global migration methods, by supporting the movement of individual and/or of multiple stream components, and by permitting such movements to be initiated by stream components themselves or by remote sites. Furthermore, splitting and merging stream components constitutes an interesting form of data migration; it addresses the computationally intensive nature of certain stream components.

The implementation of task migration is non-trivial on the distributed memory machines targeted by ACDS, especially when multiple tasks must be migrated in order to complete some desired stream adaptation. Issues to be addressed include the consistent transfer of internal task state from the source to the destination across heterogeneous machines, the rerouting of connections while dealing with messages that may still be in transit, and interfacing with heterogeneous OS platforms and their system call interfaces. ACDS approaches these issues as follows. Concerning heterogeneity, we assume that stream components offer explicit operations for state saving and restoring, which may be called by ACDS adaptations. The re-routing of connections is handled by use of buffering in stream components, with buffer management being under the control of ACDS. The current implementation of ACDS runs with the Solaris, Irix, and Linux operating systems.

The novel concept provided for task migration by ACDS is that of distributed adaptation enactment with a transaction model. This is described in more detail in Section 4.2 below.

Task Splitting. We call the set of parallel tasks generated through splitting a *program*, while its individual components are called *tasks*. Splitting is difficult when performed for a stream component (i.e., a program) that communicates with other components, each of which may itself consist of multiple tasks. To address these difficulties, the split operation may be used in three different modes.
Parallel Mode. Each copy of the source task performs a different job, with tasks negotiating for jobs. This mode is used to increase the level of parallelism of the stream component being split. Sample uses of this mode include executing the same code on different data (SPMD) or executing different codes on shared parts of the data (MIMD). Stream components split in this fashion must make the correct assumptions concerning the necessary synchronization at their respective inputs and outputs. The alternative synchronization methods supported by buffer management and communication support in ACDS stream components are explained next.

Figure 2 depicts a situation in which 'Program 2' is split into two tasks, with each task operating on half the data. Figures 3 and 4 depict three alternate synchronization methods

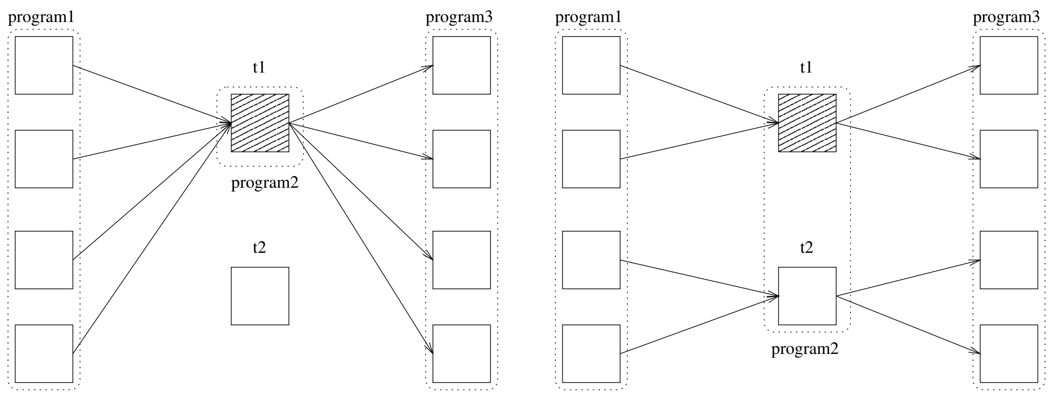

Figure 2: Start Position and Distribution Mode

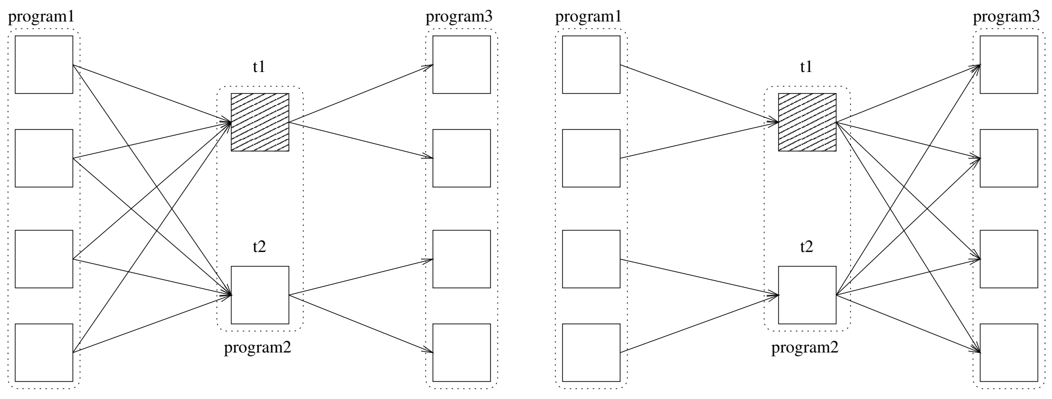

Figure 3: Global Synchronization on Task Input vs. Output

employed by the parallelized version of 'Program 2': (1) synchronization at the inputs of 'Program 2', (2) synchronization at its outputs, and (3) synchronization at both its inputs and outputs.

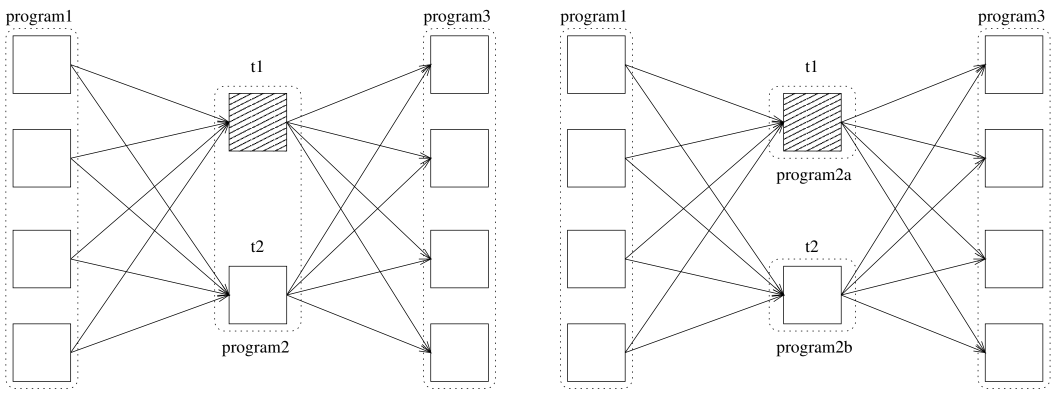

Figure 4: Global Synchronization on both Input/Output and 'Configuration Mode'

A typical use of the split operation is one in which some particular stream component is parallelized, followed by the reassembly of results in a subsequent component. Cost models resident in the MST can provide estimates of the potential benefits of split operations, as shown with the experimental results described in Section 6. More complex cost models targeting real-time applications are described in [RSYJ97].
Redundant Mode. A program 'split' in 'redundant mode' generates two parallel tasks (from one initial task) that both execute the same operations on the same data, and that send their outputs to the same target(s) as the initial task. This mode is useful when splitting is performed to improve component reliability. However, ACDS does not currently offer built-in support for voting on task outputs.
Configuration Mode. A program 'split' in 'configuration mode' again results in two identical tasks. However, their outputs may be directed at different targets, a fact that is not obvious from the depiction of such a split operation on the right hand side of Figure 4. This mode is useful when dynamically creating a 'branch' in a computational data stream, perhaps to process and visualize the same data as the original branch, but using different processing methods and displays. An example drawn from the sample application presented in Section 3 is one in which one stream branch extracts physical information from atmospheric data (e.g., wind velocities, etc.), whereas the second, new branch extracts chemistry information (e.g., ozone concentrations).

Merging. Merging is the inverse operation of splitting. The only difficulty with merging concerns connection reconfiguration for adjacent stream components. ACDS' buffer management and communication facilities integrated with stream components automatically deal with such reconfiguration. As with component splitting, a cost model is employed by the decision-making component of ACDS (ie., the MST) when making merging decisions.

4.2 Adaptation Transactions

One problem with splitting, merging, and otherwise changing stream components is that there typically exist dependencies across multiple components, as exemplified by the communications between connected components. While the anonymity of the ECHo event mechanisms helps with the implementation of communication reconfiguration, ECHo does not address dependencies across stream components that concern changes in internal component states, as exemplified by a 'split' in which half the outputs from the previous component should be sent to one vs. the other split task. The resulting need for multi-component adaptations is met by ACDS' adaptation transactions [GR93, Lam81].

ACDS' adaptation transactions constitute a distributed version of the multiprocessor mechanism first described in [BS91] and are in general a modified version of a 2-phase transaction protocol used in databases. However, ACDS adaptation transactions do not attempt to guarantee their completion times. Furthermore, our current implementation assumes a no-fault error model due to our principal interest in performance rather than reliability. Finally, ACDS transactions are not limited to directed acyclic graphs of computational data streams, but can also deal with cycles in connection topologies.
Adaptation transactions operate as follows:

- Phase 1: (1) notify all stream components involved in the adaptation step concerning the changes they must each apply, using a unique, monotonically increasing adaptation identificator 'AID' for this adaptation; (2) wait for every component to send the acknowledgment that the adaptation can be executed; (3) upon receipt of all acknowledgments, start Phase 2 of the adaptation.

- Phase 2: (1) send a message with adaptation id 'AID' to all sources of inputs of the stream component being adapted; this causes all necessary adaptations of the sources (if any); (2) when the stream component being adapted has received messages with this AID from all of its input sources, it then completes its own adaptation and executes any necessary cleanup actions; all messages sent by a component are tagged with the largest 'AID seen so far by this component; (3) for failure detection, it also sends a message to the monitoring and steering tool (MST). It is up to each stream component to decide how it should best implement its own adaptations.

- Abort: if an adaptation should be aborted while still in progress, an abort message is sent, which clears all ongoing adaptations using the same propagation principle as described in Phase 2. Adaptations are aborted when the MST determines them to be unnecessary or inappropriate, if one component cannot enact the adaptation, or when failures occur.

Additional detail on adaptation transactions, including possible optimizations, graph analysis methods and the fashion in which cycles are removed, failure recovery, and how to deal with concurrent adaptation requests appear in [Ise99].

5 ACDS: Architecture and Implementation

5.1 Framework for Stream Component Construction

Given a code module that implements the basic functionality of a stream component, it should be straightforward to construct a new stream component and integrate it into existing streams. In particular, component programmers should not have to be concerned about the underlying ACDS structure that monitors component behavior, supports splitting and merging, implements adaptation transactions, and handles component connections.

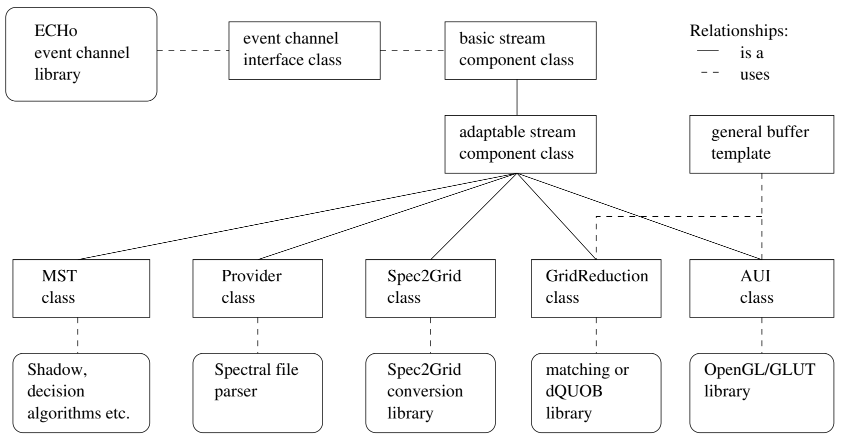
ECHo event channel library
event channel interface class
basic stream component class
Relationships:
is a
uses
adaptable stream component class
general buffer template
MST class
Provider class
Spec2Grid class
GridReduction class
AUI class
Shadow, decision algorithms etc.
Spectral file parser
Spec2Grid conversion library
matching or dQUOB library
OpenGL/GLUT library

Figure 5: Class Hierarchy and Libraries

The implementation of ACDS stream components utilizes two basic C++ classes, as depicted in Figure 5. First, the *basic stream component* class provides communication support via its *event channel interface* class and other basic utilities like monitoring and buffer management. This class uses ECHo event channels for inter-component communications. Second, the *adaptable stream component* class provides everything that is necessary to carry out adaptations, including monitoring, AID management, and adaptation enactment.

A new stream component is created by deriving a new application class from the *adaptable stream component* class, then overriding the virtual methods for computation, providing the computational code in the stubs, identifying the internal state that should be transferred in case of component migration, and finally, supplying information about the manner in which the new component may be split into multiple tasks (and merged) and about any other adaptations that are supported by this component. For the sample application described in this paper, each stream component is derived from the adaptable class, even the data provider, the AUI, and the monitoring and steering tool (MST) itself (see Figure 5).

When an element is split and then runs as two parallel copies, each supplying data to a common target, the target may have to deal with out of order data arrivals. To shield the application programmer from this problem, a standard buffering technique is provided for reassembling input data. However, if in-order delivery is not required there is no need to use these buffers. A more detailed description can be found in [Ise99].

5.2 Internal Component Structure

The internal structure of a stream component is depicted in Figure 6. This structure is not visible to application programmers, but it is useful to describe it to place into context the performance results presented in Section 6.

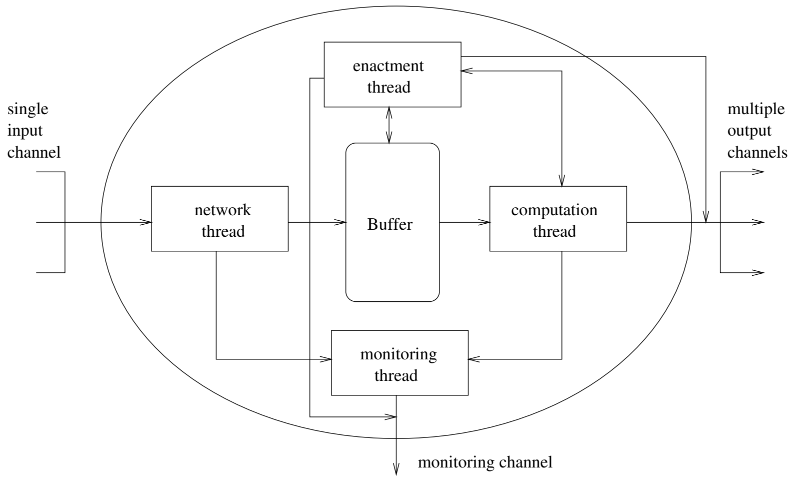

Figure 6: Adaptable Stream Component

Each stream component consists of four different threads, two of which perform most of the relevant work. The monitoring thread gathers timing information about the performance of this stream component and counts events, including incoming events, computation times, and outgoing events. This information is sent over the monitoring channel to the MST. Monitoring events can also serve as acknowledgments for completed adaptations. The enactment thread carries out adaptation transactions, and it interacts with the MST and with other stream components, as necessary. Both of these threads are run periodically.

Most of the actual 'work' in a stream component is performed by the network thread accepting inputs and the computation thread running the component's code and issuing output events. Since event communications are asynchronous, each stream component can take advantage of communication/computation overlap in its operation.

5.3 MST Structure

The Monitoring and Steering Tool (MST) supervises ACDS' stream operation and adaptation. Its main components are the data management system, adaptation decision algorithm, and adaptation enactment mechanism. Data management keeps track of the stream's task graph, of the node graph of available processors, and of the mapping of tasks to processors. Associated with each task is a monitoring trace window and other attributes like mapping constraints, available adaptation actions, and operating system. The MST also performs resource management, by keeping track of previously created local and remote processes. These processes act as 'containers' for newly created stream components and their tasks. Once created, such containers are 'acquired' in response to 'split' operations and 'released' when tasks are 'merged'.

At startup time, each stream component sends a registration message to the MST via the system's monitoring channel; the message contains application-specific information with which data management in the MST constructs its own 'shadow' of each stream component. Runtime component registration with the MST is coupled with the fact that the MST decides on changes like 'split' and 'merge', guarantees the consistency of the resulting 'view' of the computational stream maintained in the MST. The MST uses its internal view of the computational stream when executing its decision algorithms to suitable stream adaptations. A detailed description of the decision algorithms employed in the MST appears in [Ise99]. Discussions concerning the effects of monitoring rates and detail on the performance of MST decision algorithms appear in Section 6 of this paper and in more detail for real-time applications in [RSYJ97]. Currently, the main bottleneck for large data volumes is the MST as shown by experiments in [Ise99]. Our solution to this problem is also discussed there. As the MST is a stream component itself, we can apply exactly the same operations as to the other elements, namely split, merge, migrate, and parameter changes to build dynamiccly a hierarchy of MSTs.

6 Evaluation

6.1 Experiment Platform

This section utilizes output data from the atmospheric simulation described in Section 3. The measurements reported here use a cluster of 16 Sun Ultra 30 workstations (each with 128MB RAM, 247MHz, running Solaris version 2.5.1). These machines are connected via switched 100MBit/s Ethernet links. Data is displayed with an OpenGL-based visualization tool running on SGI O2 machines (each with 64MB RAM, 195MHz, running Irix version 6.3). The SGI machines are connected to the Sun Ultra cluster via 10 MBit/s Ethernet.

The atmospheric data used in our experiments is organized by simulation time steps and by the 3D nature of this data set. Specifically, each time step simulates 2 hours of real time; atmospheric data is comprised of 9 different species, each having 64 longitudes, 32 latitudes, and 37 level values, where each value is represented by a floating point number. This results in roughly 2.7MB of data per time step in grid format. For long term storage, this data is compressed into spectral form, with a resulting constant compression rate of 4.04, thereby reducing data size for one time step to roughly 675KB.

A 'debugging' model run simulates at least 6 weeks of real-time and generates a total of about 340MB of spectral data. A run used for interesting scientific inquiries might simulate 1-2 years of real-time and produces about 1.5 to 3GB of spectral data, which translates to about 12GB in grid format. Compared to other scientific applications, these data amounts are still small. Today's large data sets can easily reach the order of 1TB, and the trend for larger data sets seems to continue. The atmospheric data file used in our experiments resides on the local disk of one of the Sun machines on which the provider runs.

6.2 Application Benchmarks

ACDS' utility for high performance data streams has two sources: (1) its ability to react to changes in the availability of underlying computing resources and (2) its ability to react to changes in end user needs. This differentiates ACDS from traditional research in load balancing and migration [CX94, LK87, LMR91, BS85].

The ACDS prototype evaluated in this paper derives its knowledge about changes in end user needs from the user interface itself, by watching the end users' manipulations of the data being displayed. As pointed out in a recent publication by our group [IKS+99a], the resulting 'active user interface' (AUI) can substantially improve stream performance, by ensuring that the data being streamed is exactly what is currently needed (being viewed or manipulated) by the end users. Control events issued by the AUI are used for this purpose. We illustrate such improvements with results attained from an experiment with the computational stream described in this paper and with a simple active user interface written with the OpenGL library. The experiment is performed in a WAN environments spanning the Internet.

Approach	Request time in s
AUI	188
Traditional	2016

Table 1: Atlanta-Munich Experiment

Table 1 presents the results of this WAN experiment, where a user at Georgia Tech dynamically changes his behaviour, by viewing different portions of the atmospheric data set, the latter being generated at the TU Munich, Germany. Performance improvements are derived from reductions in the total amounts of data being transmitted (and transformed) by this computational data stream. These reductions are due to the fact that control events produced by the AUI adapt the stream to send only the data actively being viewed by the user. The times listed are the total times required to display all data actually viewed by the user in representative stream executions. Performance improvements vary depending on user actions.

The importance of this experiment is its demonstration of ACDS' ability (1) to adapt data streams in response to changes in end user needs and (2) to generate substantial end-to-end performance improvements by performing such runtime adaptations. More generally, in [IKS+99b], we demonstrate the utility of the AUI-based approach to data stream management for (1) improving query response times and (2) stream scalability with respect to both the number of clients using stream outputs and the amounts of data injected into the stream. A third advantage of runtime stream adaptation is demonstrated next, by reacting to dynamic changes in the capabilities and loads of the computational engines used by streams.

Migration experiments. By default, whenever a user connects some user interface to a running instance of the atmospheric model's data stream, the stream components necessary for data conversion (e.g., Spec2Grid) and reduction (e.g., GridReduction in response to AUI-based control events) are initially created on the host on which the user interface resides.

Better initial placements could be computed, but the main goal for our approach was to be able to cope with *dynamic* changes in data stream sizes. The main purpose of this experiment is to show that ACDS is able to react bring the system from a bad situation — e.g., due to dynamic changes — into a better one. The data provider for the stream runs on the computational cluster. In the LAN-based experiments shown next, this host is the SGI machine connected to our computational cluster of SUN workstations via a 10MB Ethernet link.

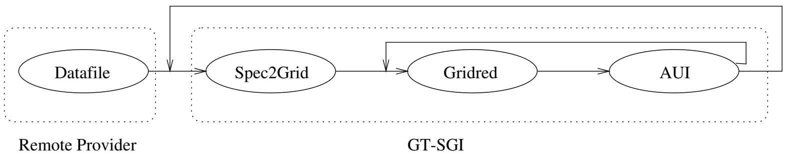

Figure 7: Initial Configuration

The resulting stream configuration is termed the 'Initial Configuration' in Figure 7 and does not exhibit good performance, principally because the SUN cluster has internal 100MBit/s connections vs. the 10MBit/s connection of the cluster to the SGI machine. The ACDS MST tool discovers this fact by monitoring stream performance. In response, it first migrates the Spec2Grid stream component to the SUN cluster, followed by the migration of the GridRed component (see Figure 8), the latter being the stream component that reduces network bandwidth needs by filtering the stream in response to changes in end user behavior seen by the AUI. The AUI itself and the DataFile provider cannot be migrated.

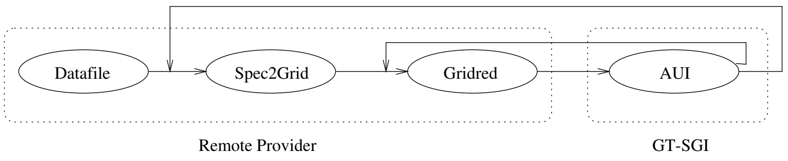

Figure 8: Situation after Migration

Table 2 presents the results for a 'debugging' model run. The first two rows represent the best and the worst cases without the MST enabled and the stream configured by hand. The times shown are the total stream execution times for both cases. The third row depicts total stream execution time when using ACDS' stream monitoring and adaptation and the decision algorithm currently embedded in the MST. Total stream execution time with MST is worse than the best case due to delays in MST's recognition of stream performance problems and due to the costs of adaptation enactment. These delays and costs are explained in more detail in Section 6.3.

These results are encouraging, since performance with MST is only 8.3% worse than the best possible performance attained by manual component placement. Specifically, these results demonstrate that the current delays and overheads due to MST usage are acceptable for the computational data streams addressed by the ACDS system.

The previous experiment demonstrated MST's ability to deal with heterogeneity in the underlying computing infrastructure. In comparison, the experimental results depicted in

Configuration	Time in s
Middleware on Sun	109
Middleware on SGI	437
Migration with MST	118

Table 2: Performance Improvements due to Migration

Configuration	Time in s
No load	109
Load on GridReduction, no migration	337
Load on Spec2Grid, no migration	328
Load on Spec2Grid node, migration	134
Load on GridReduction node, migration	151

Table 3: External Load

Table 3 concern performance improvements derived from ACDS and the MST in response to runtime changes in system loads. Specifically, in these experiments, we impose large additional loads on the respective computational engines. With a small delay due to the reasons mentioned in Section 6.3, the stream components are migrated to idle nodes, and the stream asymptotically reaches its best performance.

Dynamic stream behavior. Changes in machine loads and in user requirements are two causes of stream adaptations. A third cause are runtime variations of the execution times or the communication bandwidths due to the dynamic behavior of stream components themselves. Such dynamic behaviors are common in complex components with many internal branches taken in response to the data values received as inputs.

The experiment described next simulates such component behavior, by varying the computation time of the 'Spec2Grid' component in relation to the types of atmospheric species being transformed. For experiment purposes, we assume that the most 'expensive' species requires 30 times the execution time of the 'normal' species. Runtime changes in computation times are due to users' selections of the species being viewed.

This experiment demonstrates the utility of the 'split' operation on stream components, where a user's switch from the normal to the expensive species results in a component split and therefore, in the reduction of stream component execution time due to parallelization. Figure 9 depicts the situation after the Spec2Grid element has been split once and when the expensive species is being transformed. The resulting performance improvements are discussed next.

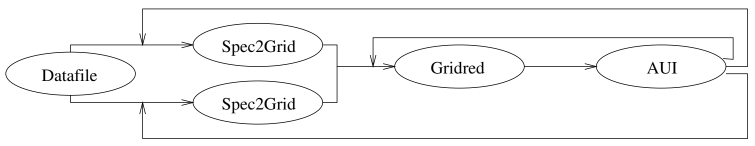

Figure 9: Situation after Splitting Spec2Grid

Figure 10 shows the speedup graph for Spec2Grid. Speedup is good for two tasks, but it is not present for three or four tasks. This lack of speedup is due to workload imbalances across the split tasks.

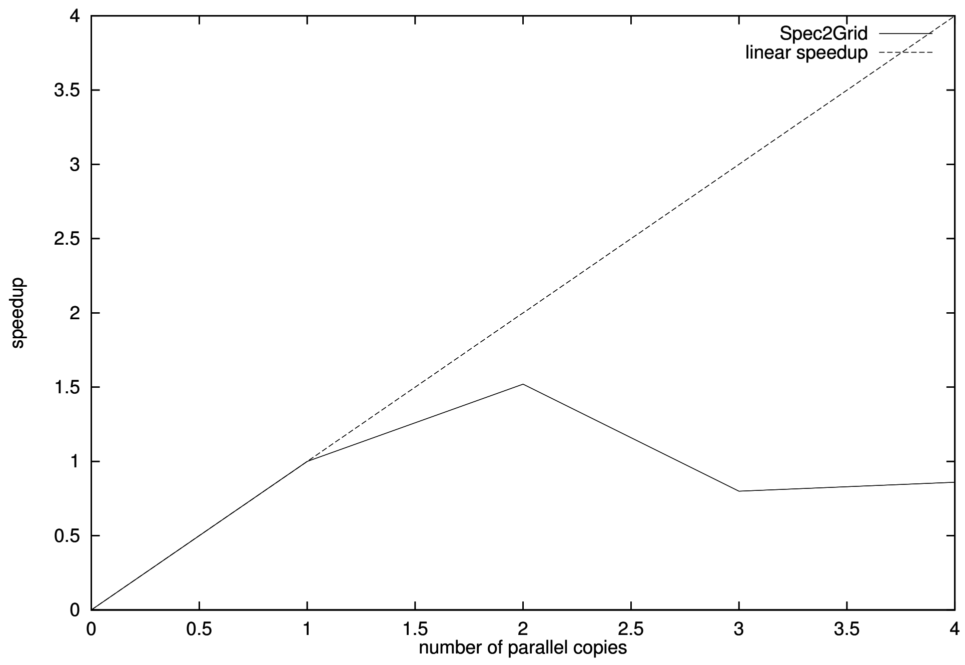

Figure 10: Speedups for Parallel Spec2Grid

The current MST tool does not have sufficient information about stream components to determine optimal levels of parallelism. Instead, it must determine suitable stream configurations by trial and errror. The resulting stream performance for one sample data run is depicted in Table 4. In this run, the computationally expensive species is requested for 100 time steps. Four different scenarios are measured. In the first scenario, the stream is not adapted at all, so that a single instance of Spec2Grid is used throughout. In the second scenario, the stream always uses two instances of Spec2Grid. An inappropriate stream configuration, using three instances of Spec2Grid, is shown in the third row. Finally, performance for the same run with MST enabled appears in the fourth row. Throughout this run, the level of parallelism for Spec2Grid varies from 2 to 3, resulting in repeated split and merge operations. More sophistical cost models of computational streams and therefore, more appropriate runtime adaptations on streams are under current development for the MST tool (see [Ise99] for further discussions of this topic).

6.3 Enactment Costs

This section describes some of the delays and overheads inherent in the current implementation of ACDS and its MST tool (complete descriptions appear in [Ise99]). ACDS uses the 'rsh' command to start up remote processes. This results in large delays for split operations, in the range of $1.3s$ - $5s$. Furthermore, the initial establishment of and also subscription to

Configuration	Time in s
Single instance of Spec2Grid	127
Two instances of Spec2Grid	84
Three instances of Spec2Grid	157
Adaptations turned on	115

Table 4: Performance Improvements due to Splitting

communication channels remains slow, due to the lack of optimization for these actions in the current ECHo channel implementation, resulting in startup times for 'connected' tasks ranging from $2.9s$ to $6.2s$.

More interestingly, the enactment of adaptation decisions, that is, the execution of adaptation transactions, typically takes only a few seconds, thereby making it feasible to adapt computational data streams with delays suitable for end users operating user interfaces via a keyboard or a mouse. Additional performance improvements are required for immersive interfaces. Specifically, we have observed average adaptation enactment times of $1.5s$ in our LAN environment, with actual times ranging from $0.88s$ to $3.79s$. Accordingly, the current MST implementation is limited to performing adaptations at a rate of no more than 1 per $4s$.

7 Conclusions and Future Work

The main contribution of this paper is the implementation, design, and evaluation of a framework for constructing and adapting computational data streams (ACDS). ACDS is used to automatically and dynamically configure an access grid computation represented as a computational data stream. Such a stream is typically associated with a parallel/distributed scientific computation that generates stream inputs or consuming stream outputs.

Dynamic stream configuration is performed in response to changes in end user needs and in underlying machine performance. Monitoring data is gathered via control events associated with ACDS computational data streams. Configuration decisions are made by decision algorithms embedded in a stream controller, called the Monitoring and Steering Tool (MST). ACDS provides explicit support for several configuration actions on streams, including parameter changes local to individual stream components, component migration, splitting, and merging. Configuration actions are carried out by adaptation transactions that use a specialized commit/abort transaction protocol.

Computational data streams are constructed using a C++ library comprised of classes that offer the necessary functionality of ACDS stream components. Using these classes, developers simply provide the application-specific functions embedded in each component. If components are to be adaptive, developers must also provide methods to extract internal monitoring state and possibly, additional methods that adapt the component's internal operation. Additional functionality developed by our group to facilitate component monitoring and steering is described in [ES98].

Experimental results presented in this paper demonstrate the utility of runtime stream adaptation, in a WAN environment in response to changes in end user needs, and in a LAN environment in response to changes in user needs and/or underlying system (network or CPU) loads. Specific adaptations evaluated in this paper and demonstrated useful for computational data streams include component migration, splitting, and merging.

Ongoing work with ACDS includes improvements in the rate at which adaptations on streams may be performed and improvements in the algorithms that make adaptation decisions. Future research should address the scalability of systems like ACDS to the large-scale, wide area 'access grid' computations and 'portals' now being envisioned by HPC researchers. Specific topics include the hierarchical structuring of system monitoring and steering methods and tools, additional support for system reliability, and the integration and use of multiple system and network monitoring tools, including MOSS [ES98] or OMIS [LWSB97] for individual stream components, NSR [RLB98] for node monitoring, and ReMoS [DGL+97] for network monitoring.

Acknowledgements. Thanks to Greg Eisenhauer, Vernard Martin, and Beth Plale for their ideas and their support for this project. Thanks to Arndt Bode and Thomas Ludwig from the TU Munich for their support of this work at Georgia Tech.

References

[BS85] A. Barak and A. Shiloh. A distributed load-balancing policy for a multicomputer. *Software - Practice and Experience*, 15(9):901–913, September 1985.

[BS91] Thomas E. Bihari and Karsten Schwan. Dynamic adaptation of real-time software. *ACM Transaction on Computer Systems*, 9(2):143–174, 1991.

[CX94] Francis Lau Chengzhong Xu. Decentralized remapping of data parallel computations. In *Proceedings of Scalable High-Performance Computing Conference*, pages 414–421, 1994.

[DGL+97] Tony DeWitt, Thomas Gross, Bruce Lowekamp, Nancy Miller, Peter Steenkiste, Jaspal Subhlok, and Dean Sutherland. ReMoS: A resource monitoring system for network-aware applications. Technical Report CMU-CS-97-194, Carnegie Mellon, 1997.

[Eis99] Greg Eisenhauer. The echo event delivery system. Technical Report GIT-CC-99-08, College of Computing, Georgia Institute of Technology, Atlanta, GA 30322-0280, 1999.

[ES98] Greg Eisenhauer and Karsten Schwan. An object-based infrastructure for program monitoring and steering. In *Proceedings of the 2nd SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT'98)*, pages 10–20, August 1998.

[FK97] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *Intl J. Supercomputer Applications*, 11(2):115–128, 1997.

[GES98] Weiming Gu, Greg Eisenhauer, and Karsten Schwan. Falcon: On-line monitoring and steering of parallel programs. In *Concurrency: Practice and Experience*, volume 10, pages 699–736, 1998.

[GKP97] G. A. Geist, J. A. Kohl, and P. M. Papadopoulos. CUMULVS: Providing fault-tolerance, visualization and steering of parallel applications. *International Journal of High Performance Computing Applicatio ns*, 11(3):224–236, 1997.

[GR93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[GW96] A. S. Grimsaw and Wm. A. Wulf. Legion – a view from 50,000 feet. In *Proceedings of the Fith IEEE International Symposium on High Performance Distributed Computing*, August 1996.

[HK98] J. K. Hollingsworth and P. J. Keleher. Prediction and adaptation in active harmony. In *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing*, 1998.

[HS94] P. Homer and R. D. Schlichting. A software platform for constructing scientific applications from heterogeneous resources. *Journal of Parallel and Distributed Computing*, 8(9):301–315, June 1994.

[IKS$^+$99a] Carsten Isert, Davis King, Karsten Schwan, Beth Plale, and Greg Eisenhauer. Steering data streams in distributed computational laboratories. In *High Performance Distributed Computing (HPDC8)*, 1999. full report available as GIT-CC-99-12.

[IKS$^+$99b] Carsten Isert, Davis King, Karsten Schwan, Beth Plale, and Greg Eisenhauer. Steering data streams in distributed computational laboratories. Technical Report GIT-CC-99-12, Georgia Institute of Technology, 1999.

[Ise99] Carsten Isert. A framework and adaptive methods for improving the performance of dynamic parallel programs. Master's thesis, Technische Universität München and Georgia Institute of Technology, 1999.

[JRZ$^+$97] Yves Jean, William Ribarsky, Song Zou, Jeremy Heiner, Karsten Schwan, and Bobby Sumner. Collaboration and visual steering of simulations. In *Proceedings of the SPIE Conference on Visual Data Exploration and Analysis IV*. SPIE, Feb. 1997.

[KSS$^+$96] Thomas Kindler, Karsten Schwan, Dilma Silva, Mary Trauner, and Fred Alyea. A parallel spectral model for atmospheric transport processes. *Concurrency: Practice and Experience*, 8(9):639–666, November 1996.

[Lam81] B. W. Lampson. Atomic transactions. In B. W. Lampson, M. Paul, and H. J. Siegert, editors, *Distributed Systems — Architecture and Implementation*. Springer Verlag, 1981.

[LK87] F. C. H. Lin and R. M. Keller. The gradient model load balancing method. *IEEE Transactions on Software Engineering*, pages 32–38, 1987.

[LMR91] R. Lüling, B. Monien, and F. Ramme. Load balancing in large networks: A comparative study. *Proc. of the 3rd IEEE Symposium on Parallel and Distributed Processing (SPDP'91)*, pages 686–689, 1991.

[LTBL97] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny. Checkpoint and migration of unix processes in the condor distributed processing system. Technical Report 1346, University of Wisconsin, April 1997.

[Lud92] Thomas Ludwig. *Lastverwaltungsverfahren für Mehrprozessorsyteme mit verteiltem Speicher.* PhD thesis, Technische Universität München, 1992.

[LWSB97] T. Ludwig, R. Wismüller, V. Sunderam, and A. Bode. *OMIS — On-Line Monitoring Interface Specification (Version 2.0)*, volume 9 of *Research Report Series, Lehrstuhl fr Rechnertechnik und Rechnerorganisation, (LRR-TUM), Technische Universitt Mnchen.* Shaker, Aachen, 1997.

[MPI] The message passing interface standard documents. http://www.mpi-forum.org/docs/docs.html.

[MS98] Vernard Martin and Karsten Schwan. ILI: An adaptive infrastructure for dynamic interactive distributed applications. In *Proceedings of the Fourth International Conference on Configurable Distributed Systems*, pages 224 – 231. IEEE, IEEE Computer Society, May 1998.

[RD97] R. Preis R. Diekmann, B. Monien. Load balancing strategies for distributed memory machines. In Karsch/Monien/Satz, editor, *Multi-Scale Phenomena and their Simulation*, pages 255–266. World Scientific, 1997.

[RKC99] Manuel Roman, Fabio Kon, and Roy H. Campbell. Supporting dynamic reconfiguration in the dynamicTAO reflective ORB. Technical Report UIUCDCS-R-99-2085, University of Illinois at Urbana Champaign, February 1999.

[RLB98] C. Röder, T. Ludwig, and A. Bode. NSR – A Tool for Load Measurement in Heterogeneous Environments. In A. Bode, A. Ganz, C. Gold, S. Petri, N. Reimer, B. Schiemann, and T. Schneckenburger, editors, *Anwendungsbezogene Lastverteilung — ALV'98*, pages 133–144. Technische Universitt Mnchen, Februar 1998.

[RSYJ97] Daniela Ivan Rosu, Karsten Schwan, Sudhakar Yalamanchili, and Rakesh Jha. On adaptive resource allocation for complex real-time applications. In *18th IEEE Real-Time Systems Symposium, San Francisco, CA*, pages 320–329. IEEE, Dec. 1997.

[S$^+$99] Rick Stevens et al. Plans for creating access grid technology for the national machine room. TeamB/C Working Group Meeting Minutes, Argonne National Laboratories, Chicago, February 1999.

[WS99] R. West and K. Schwan. Dynamic window-constrained scheduling for multimedia applications. In *IEEE International Conference on Multimedia Computing and Systems*, 1999.

[ZLP95] Mohammed Javeed Zaki, Wei Li, and Srinivasan Parthasarathy. Customized dynamic load balancing for a network of worksations. Technical Report 602, The University of Rochester, December 1995.

[ZML99] Victor C. Zandy, Barton P. Miller, and Miron Livny. Process hijacking. submitted for publication, February 1999.

[ZS99] Dong Zhou and Karsten Schwan. Adaptation and specialization for high performance mobile agents. In *COOTS 99*, 1999.