ILI: An Adaptive Infrastructure For Dynamic Interactive Distributed Applications *

Vernard Martin   Karsten Schwan
College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332
e-mail: {vernard, schwan}@cc.gatech.edu

Abstract

This research addresses the middleware necessary to realize the construction of distributed laboratories in which scientists can construct and use complex computations and then analyze and share their results across geographically separated collaborations. We present a model of such middleware for heterogeneous distributed computing platforms and also show how middleware can dynamically adapt to changes in behavior of instruments and end users.

1. Motivation

There is a new class of applications characterized by heterogeneous distributed environments in which high performance applications interact with multiple users, visualizations, storage engines, and I/O engines. We term such an environment a distributed, computational laboratory. In such a laboratory, scientists collaboratively employ large scale computational instruments and also analyze and share results with geographically separated colleagues.

The components of a distributed laboratory have properties that differ in substantial ways from those of heterogeneous high performance computations and therefore, they demand a new set of services. For instance, while communication infrastructures like MPI [5] support jointly compiled application components on heterogeneous systems using reserved computing and network resources, the distributed laboratory infrastructure should support frequent dynamic client arrivals and departures, variable end user needs, and separately developed components, where agreement on such issues as precise data forms may be difficult to achieve.

[1]This work was funded in part by NSF equipment grants CDA-9501637 and CDA-9422033, DARPA grant ECS-9411846, Honeywell grant B09333218 and DARPA grant DAAH04-96-1-0209.

To address the demands of applications executing in a distributed laboratory, we have developed an infrastructure – called ILI (Interactivity Layer Infrastructure) – that can dynamically adapt to changes in the behavior of the computational instruments and end users across a shared and dynamically changing set of computational nodes and networks.

The remainder of this paper first describes a sample scenario laboratory; followed by a presentation of a model of the infrastructure and the basic architecture needed to support this functionality. The current ILI prototype's experimental evaluation appear in Section 6. Related research and conclusions are discussed last.

2. A Distributed Laboratory for Atmospheric Modeling

A typical distributed laboratory is depicted in Figure 1. In particular, consider a large scientific simulation running on a set of computational resources, such as our global climate transport model (GCTM) [9]. This model along with local climate, atmospheric or pollution models runs concurrently on a variety of parallel and distributed computing resources. Model outputs are processed by a variety of instruments, including specialized visualization interfaces or computational instruments performing calculations which derive from and expand upon the basic model results. Similarly, model inputs may be provided via other instruments that either utilize live satellite feeds or access stored satellite data on remote machines.

Consider the introduction of multiple observers into this application scenario. The scientists' interests may vary from wishing to see the "big picture", to investigating in detail subsets of the simulation's output, to collaborating with one another via the computational instruments these models implement. To address such needs, the distributed application has additional components that assemble the in-

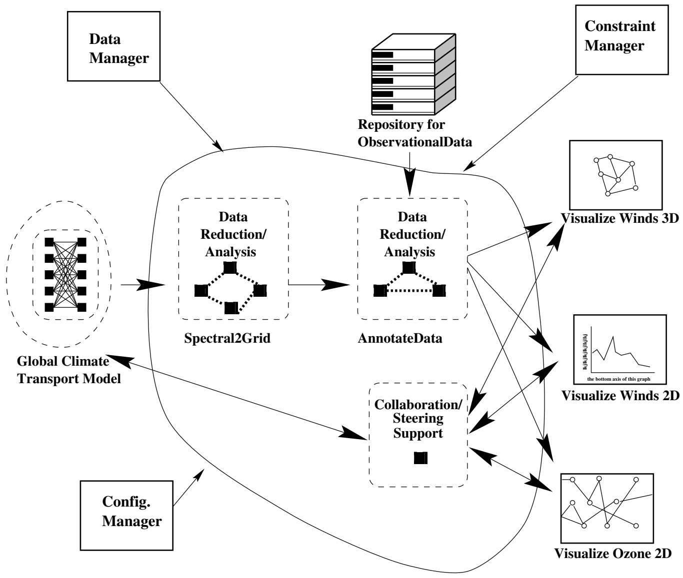

Figure 1. Sample Application and ILI High Level Architecture

formation needed to drive various interactive displays, by gathering data from the distributed simulation and performing the analyses and reductions required for these displays. Some of these components may themselves have substantial computational or storage needs and require dedicated, additional resources of their own. They may also require access to additional information, as in the case of the atmospheric model's display with which end users compare observational (satellite) data with model outputs in order to assess model validity or fidelity. In summary, since end users control the set of computational instruments, input and output components will change dynamically, driven by end users' needs or by the current needs of the running simulation.

A representative dynamic behavior is one in which an atmospheric researcher verifies the model's accuracy by validating the transport portion of the model against a known database of values. In order to accomplish this, the scientist instruments the model so that it generates monitoring data representing the current wind velocity and/or ozone mixing ratio at each timestep of the simulation. The scientist then sets up a chain of tasks to process the monitoring data for eventual 2D and 3D visualizations of the results. The chain of tasks is necessary, because the data generated by the model is in spectral form and must be converted to grid form for viewing by the VisualizeWinds2D instrument.

When using the VisualizeWinds2D for validation, a high level of detail in the data is required, which prompts the scientist to increase the rate of monitoring data so as to gain an improved time resolution. If after a few minutes of observation, the scientist decides to view the data in 3D, the VisualizeWinds3D instrument is started, thereby enabling simultaneous 2D and 3D views.

The dynamic behaviors described above are addressed by our ILI framework, which (1) supports dynamic components attachment to distributed simulations, and (2) captures the real-time nature of component interactions. More precisely, ILI supports task creation and deletion, changes in information flow between tasks, and it offers QoS specifications for tasks and flows, and heuristics that attempt to meet such QoS specifications.

3. ILI: An Interactivity Layer Infrastructure Model

Model Components: ILI models its 'applications' as sets of *tasks*, T_1 to T_n, communicating via *events*. Each task has a set of input events, I_1 to I_n, required to perform its work. When all necessary input events have been received, its computation is triggered. Each task may access and alter some internal state, *S*, while performing its computation. Based on its trigger and its internal state, a task fires and then generates some set of output events, O_1 to O_n.

For the distributed laboratory scenario discussed previously, the global climate model written with a computing infrastructure like MPI is treated as a set of tasks known to ILI, as are all of the instruments used in conjunction with this model are also represented as ILI tasks. The model tasks have no input events, but produce output events in spectral form describing wind field values and ozone mixing ratios. In this case, there are up to 37 such tasks (one per atmospheric level), each producing one output event per simulation timestep. For simplicity, ILI considers them to be a single model task. The Spectral2Grid task has one input event, which is the same as the output event of the model. It also has one output event, which is the ozone mixing ratio and wind field data in grid form. The AnnotateData task has one input, which is the grid data from the Spectral2Grid task, and it produces one output event, a modified version of the grid form data for the ozone mixing ratio and the wind fields. The VisualizeWinds task has one input event, the annotated data from the AnnotateData task, and it does not produce any output events.

Management of Infrastructure State: The flow of events from task to task represents task linkages. A *task linkage graph* is a group of *nodes* and associated *directed edges* where there exists one node per task currently executing. Edges in the task linkage graph are directional. If at runtime, a particular task, T_i, sends an event to another task, T_j, then we say there is a *link* from T_i to T_j, and an edge connects task T_i to task T_j. Multiple edges may exist between two tasks. The task linkage graph encapsulates the actual run-time connectivity of tasks.

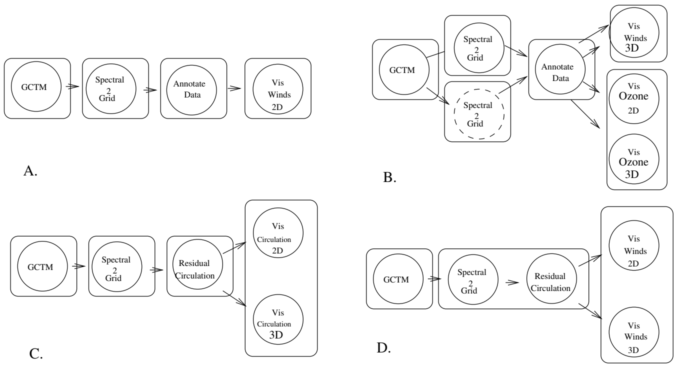

Figure 2. Evolution of task linkage graph

Figure 2 shows the evolution of the task linkage graph for the distributed laboratory GCTM verification process described in Section 2. Circles represent tasks, rounded boxes represent computational units. Dotted circles represent cloned tasks. A shows the initial TLG for the GCTM validation scenario. B shows the TLG after the addition of several visualization clients and a 'task clone' adaptation. C shows the TLG after a 'task merge' adaptation and the termination of several visualization clients. Note that the task linkage graph changes as the executing tasks change.

ILI adaptation heuristics focus on the chain-like structures linking data generation with consumers. Towards this end, a *path* is defined as a sequence of tasks, T_1 to T_n, such that for each task T_i there exists at least one output port with an event type that matches an input port for Task T_{i+1}. Note that a set of tasks may have many potential paths that are not actually realized during execution. For the distributed laboratory scenario, one such path is GCTM ⇒ Spectral2Grid ⇒ AnnotateData ⇒ VisualizeWinds.

A *computational unit* is an entity capable of executing tasks, such as a single workstation or a cluster of processors acting as one unit. The mapping of tasks to computational units is based on the concept of a *workgroup*, where each task is a member of at most one workgroup and each workgroup is a set of tasks that form a path. For this paper's sample distributed labs scenario, we define three workgroups: Workgroup 1 consist of the model, Workgroup 2 consists of the Spectral2Grid and the AnnotateData tasks, and Workgroup 3 consists of the Visualize Winds tasks.

Workgroups are mapped to computational units such that QoS specifications are met. Task to workgroup assignments may be constructed dynamically or re-assigned as task linkage graphs change. Specifically, a *work group mapping* is defined as a set of pairs (*workgroups*, *computation units*). This mapping determines the tasks that are currently assigned to execute on computational units. It also constitutes a partial schedule in that a task may not execute on a computation unit unless its *workgroup* is currently mapped to that unit. Currently, a new work group mapping is calculated only when new computational units are added or removed from the available resources.

Finally, an *attribute* is a (name,data) tuple. Attributes may be attached to *tasks*, *workgroups*, *the task linkage graph*, etc. Several *default attributes* are used for the current QoS specifications implemented by ILI; some of which are shown in Table 1. Events are represented by *e* and tasks by *t*.

$Min_{t,e}$, $Max_{t,e}$	QoS rate for event on a task
$Rate_{t,e}$	Current rate for this event with this task
$History_{t,e,n}$	Last n rates for this event with this task
$Options_t$	The adaptations available for the task
$V_{t,e}$	violation for this task for this event
δ_t	task attribute update period
Δ_t	violation checks period
L_{cu}	Load on a computational unit

Table 1. Default attributes used in ILI QoS

Default attributes are used to define the *constraint violation value*, V, which is the the magnitude of the difference between the current rate and the actual QoS min or max attribute. V_{T_i} is computed as the sum of the *constraint violation values* for each input and output event of task T_i. It is possible for a violation in one task to cause a violation in a downstream task. Propagated violations, called *external violations*, are maintained separately from a task's *internal violations*. The total violation value of a node is the sum of its internal and external violations. External violations occur when a minimum input rate is not met.

The nodes of the *task linkage graph* are annotated with these violation values as weights, thereby forming a *constraint violation graph*. *V* for each node represents its total contribution to the violations in the entire system; it is recomputed every Δ intervals.

The minimum and maximum event rates for the input and output events of each are used to determine whether or not a particular path is viable in the infrastructure. Viability permits the pruning of possible paths prior to the use of the ILI application.

Configuration Specification: The initial configuration of the infrastructure is represented as a graph. The user specifies the initial task set, which is then translated to ILI nodes. The user also specifies task input and output events and their types. A task's event data is used to construct the connections in the task linkage graph. An initial mapping to computation is specified, as well. For the GCTM verification pipeline described earlier, the specification of the initial pipeline would be **initial-config ("Transport,Spectral2Grid,Residual-**

Circ,AnnotateData,Viz","1,2,3,3,4"), which describes a one-to-one mapping of workgroups to tasks. Tasks mapped to the same number reside in the same workgroup, whereas tasks with different numbers must reside in different workgroups.

Input and output events are specified for each task. For example, the call **define-task (self,NULL,"spectral-data")** and **define-task ("Spectral2Grid","spectral-data","grid-data")**, when executed by the GCTM, defines and registers itself and one other task with the ILI. This allows one task to initiate a series of other tasks for which it will generate or receive data.

4. The ILI Architecture

The ILI infrastructure provides a set of services: data management, constraint management, and configuration management.

Data Management: ILI provides a means of classifying task data and routing it to other tasks that are interested in receiving this data [3]. Additionally, it provides a name domain for data components and tasks, maintains knowledge about and manages communication links between tasks, and it maintains a body of knowledge about all connections from tasks to the infrastructure as well as all connections internal to the components of the infrastructure.

Data Management functionality is encapsulated in an ILI entity called the Data Manager. The Data Manager enables tasks running on heterogeneous machines to understand and convert each others' data formats. In addition, the Data Manager is responsible for forwarding any events produced by tasks to those tasks that are interested in receiving them.

The Data Manager is a logical entity and does not have to be physically implemented as a single task. It may be a coordinating group of entities, or its functionality may be integrated into the tasks themselves.

For the GCTM verification scenario, the Data Manager stores the names of the individual task components that are passed to it via the API. For example, **define-self ("Transport")** executed by a task informs the Data Manager that the task used the name 'Transport'.

After a task registers itself, it may define additional tasks, each time specifying task name, its input events and its output events. Event names are registered dynamically.

Constraint Management: ILI maintains knowledge representing certain requirements of (1) the tasks in the infrastructure and (2) the data that is sent and received by infrastructure tasks. ILI also stores restrictions on both the connections to the infrastructure and the data that is sent to it. These restrictions are called *constraints* and are based on user-defined characteristics associated with data or connections. This aggregate functionality is encapsulated in an entity called the Constraint Manager. Constraints are inferred from the attributes associated with data or connections. Furthermore, the Constraint Manager maintains a history of ILI configurations as they occur in order to identify potential repetition in sequences of configurations. Simple pattern matching determines such repetitions resulting in modifications to the Configuration Manager.

In the GCTM verification scenario, we express the constraints as minimum and maximum rates for the input events consumed by and the output events produced by each task. For example, **define-constraint ("Spectral2Grid","grid-data",10,20)** and **define-constraint ("Spectral2Grid","spectral-data",-1,-1)** define a constraint for the input and output of the Spectral2Grid instrument, respectively.

Configuration Management: The infrastructure maintains knowledge about the mapping of tasks to workgroups to computational units. It also provides methods for modifying that mapping. This functionality is encapsulated in an entity called the Configuration Manager. The Configuration Manager handles task creation and deletion, as well.

For the GCTM verification scenario, the Configuration Manager is responsible for instantiating the necessary tasks when a **define-task** call is made. In addition, it receives information from the Constraint Manager informing it of a necessary reconfiguration should be done, the actions necessary for enacting it, and the desired new configuration.

Implementation Substrate: The implementation of ILI employs a substrate that provides basic event transport and management tools including: (1) routing of events from generating to receiving tasks, (2) conversion of data across heterogeneous platforms, and (3) dynamic connectivity between transient applications.

The substrate consists of: (1) event types cataloged and translated with PBIO [2], which supports event format and translation across heterogeneous machines with improved performance compared to SDDF [1], (2) events transported using the DataExchange communication infrastructure detailed in [3], (3) information flow, task execution rate, and task execution timings monitored using an on-line monitoring facility (examples of such are W3 [7], Chaos-MON, and Falcon [4]) and (4) an on-line steering facility [4],[13] allowing ILI to affect the operation of its tasks.

5. Heuristics

ILI's on-line adaptation heuristics are applied cyclically in three different steps: (1) Detect State Phase which determines if all tasks are currently meeting their QoS constraints and if not, where the violations are, (2) Predict Next State Phase which determines what reconfiguration should take place to eliminate violations, and how to change the system to a state with fewer or no violations, and (3) Shift State

Phase which performs the actual reconfiguration of the system from one state to another.

Detection of Current State: In the *constraint violation graph*, the sum of the weights of the nodes along any *path* from task T_i to task T_j represents the total extent of the violation of QoS constraints on that path. In addition, the node with the largest magnitude along that path exhibits the largest violation. By sorting by magnitude, it is possible to identify the node with the largest violation as the *offending node*. If there are multiple such nodes, the one with the least number of total incoming and outgoing edges in the corresponding task graph is selected, in order to minimize the extent of modification to the linkage graph, if possible.

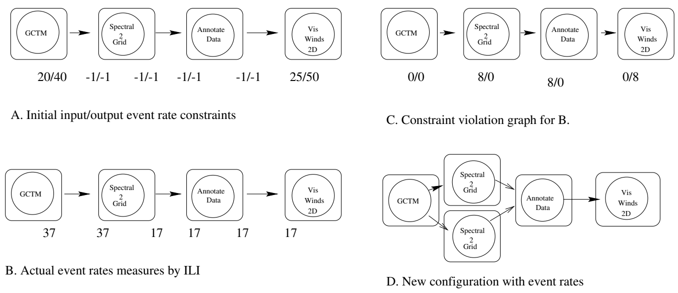

Figure 3. Constraint Violation Graph

Consider Figure 3. A shows the initial event rates specified by the constraints. B shows the event rates after Δ time. C shows the constraint violation graph that corresponds to B. Note that the Spectral2Grid and the AnnotateData instruments inherit internal violations from the external violations of the task downstream to them. As a result, the task with the largest violation is the Spectral2Grid instrument.

Predict Next State: Once it has been determined that reconfiguration is necessary, a new configuration is computed by a rate determined by the attributes δ and Δ. Toward this end, the detection heuristic identifies the target node it suspects to be the worst offender in contributing to the level of violations in the system. There are four reasons for such violations: the node is sending (1) too many or (2) too few events downstream, or it is receiving (3) too many or (4) too few events from upstream. To reduce the offending node's violation value, its adaptation options are considered. For each such option, a new constraint graph is constructed with the violation value of the offending task reset to zero, thereby representing that the violation represented by the task itself was relieved. The available adaptation options are *migrate*, *clone*, and *merge*. The migrate option permits a task to migrate left or right along a path, resulting in two new constraint graphs. A task is constrained to migrate only along its path so that it can at most move to one different workgroup, thereby creating at most one alternative constraint graph. The clone option permits a task to duplicate itself. Tasks upstream from a cloned task divide their outputs among each of the cloned tasks. Likewise, the clones feed all of their outputs to the same source as the original task. The clone adaptation creates only one alternative constraint graph. After building all potential new constraint graphs, the adaptation option producing the lowest overall violation value for the entire graph is selected and enacted. This may cause the assignments of tasks to workgroups to change.

For the GCTM scenario, the Spectral2Grid instrument is the offending task. The adaptation options available to it are clone and migrate. When computing the two possible new constraint violation graphs, the clone adaptation option is determined to be the best choice.

To prevent thrashing among configurations, the occurrence histogram maintained by the Configuration Manager is marked to identify the configuration that occurs most, and is then disallowed as a viable configuration for future selections.

The workgroup to computation unit mapping algorithm is straightforward and relies on the adaptation options for meeting QoS constraints rather than frequently shifting workgroups among computation units. The initial mapping of tasks to workgroups is done by the user.

Shift State: Once a new configuration has been determined, the system enacts the changes necessary to transform the current to the new configuration. The reconfiguration process occurs in two phases. The first phase involves sending reconfiguration messages to those tasks that either send messages to or receive messages from the target task. The second phase is the actual physical reconfiguration of communication channels and the movement of tasks to different computation units, if necessary.

The configuration manager is informed by the constraint manager that certain reconfiguration actions are required. The configuration manager then sends a 'start reconfiguration' message to the first task in the path that is directly involved in communicating with the offending task. This message includes a list of the affected tasks so that each subsequent task can inspect the message to see if it is involved. If affected, a task performs the appropriate reconfiguration, send a 'done' message to the reconfiguration manager, and then forwards the reconfiguration message downstream After the reconfiguration message is finished, the task sends a 'reconfiguration done' message to the reconfiguration manager. When a task receives a reconfiguration done message, it sends an acknowledgment to the configuration manager.

6. Evaluation of the ILI Framework

Performance Evaluation: static case The base performance of the computational instruments used in the ILI evaluation is established by encapsulating each instrument as a separate process and then configuring these processes as GCTM ⇒ Spectral2Grid ⇒ AnnotateData ⇒ VisualizeWinds2D. Each instrument, except for the GCTM itself, is assigned to a uniprocessor SGI Indy workstation connected via 10Mb switched Ethernet. The GCTM runs on a 12-processor SGI Powerchallenge connected via ATM to an Ethernet switch. The values in the tables below represent the input and output event rates of specific instruments. An output event rate that is consistently greater than the input event rate indicates that the instrument currently does not have sufficient cpu resources to process all of the events it receives. An input event rate that initially exceeds the output event rate but eventually decreases to be less than the output rate indicates that the instrument is 'catching up' on previously buffered events. The VisWinds rate represents the rate at which it is able to visualize events since this instrument does not produce events.

Monotonically increasing rate Event Rate for Instruments (in/out)				
Time	GCTM	Spectral 2Grid	Annotate Data	VisWinds 2D
5min	74(1x)	74/74	74/74	74/74
10min	148(2x)	148/120	120/120	120/120
15min	222(3x)	222/100	100/100	100 100
20min	296(4x)	296/97	97/97	97/97

Cyclic/oscillating rate (2x) Event Rate for Instruments (in/out)				
Time	GCTM	Spectral 2Grid	Annotate Data	VisWinds 2D
5min	74(1x)	74/74	74/74	74/74
10min	148(2x)	148/120	120/120	120/120
15min	74(1x)	74/74	74/74	74/74
20min	148(2x)	148/120	120/120	120/120

Table 2. Event Streams for Static Chains

Event Rate for Instruments (in/out)					
Time	GCTM	S2G1	S2G2	Annotate Data	VisWinds 2D
			–		
5min	74(1x)	74	–	74	74
10min	148(2x)	74	74	110	110
15min	222(3x)	110	110	220	220
25min	296(4x)	110	110	220	220
30min	370(5x)	100	100	200	200

Table 3. Event Streams for Clone Adaptation

The normative results for a static chain of instruments with steady event streams are shown in Table 2. These results indicate that the Spectral2Grid instrument is the bottleneck in this configuration. As the GCTM sends more data to the Spectral2Grid instrument, it becomes overwhelmed and cannot process events at the same rate as it receives them. Eventually, it spends all of its time receiving events and not actually processing them, therefore causing all instruments downstream from it to starve and do no useful work. If the amount of buffer space in the instrument is static, then the instrument may terminate abnormally as well.

Event Rate for Instruments (in/out)				
Time	GCTM	S2G	AnnotateData	VisWinds
5min	74(1x)	74	74	74
10min	148(2x)	74	110	110
	GCTM	—	S2G & Annotate	VisWinds
15min	222(3x)	—	150	150
25min	296(4x)	—	270	270
30min	370(5x)	—	370	370

Table 4. Event Streams for Migrate Adaptation

Performance Evaluation: dynamic case If the 'clone' adaptation option is applied to the Spectral2Grid instrument in the static scenario, the values presented in Table 3 result. The effect of the cloning option is that the Spectral2Grid instrument is able to process more events and forward them downstream. This particular case assumes that another relatively unloaded computation unit is available for the instrument's clone. If this is not the case, the migrate adaptation option may be used. Table 4 shows the results of this option on the Spectral2Grid instrument. In this case, the Spectral2Grid instrument is migrated to the same computation unit as the AnnotateData instrument. Once there, it is able to process more events. When the system reaches a state where there are no violations, reconfiguration will cease.

Although not explored here, the merge adaptation is also available. Merge is the reverse of cloning and effectively transfers the event processing load of one instrument to an identical instrument located elsewhere. This option occurs when an instrument consistently does not meet its minimum event threshold and there exists another identical instrument that is at no more than 75 percent of its maximum event rate threshold.

Pre-Replicated Tasks			On-Demand Tasks		
Option	time	state	Option	time	state
migrate()	15ms	2K	migrate()	1.5s	2K
clone()	38ms	2K	clone()	9.0s	2K
merge()	25ms	2K	merge()	5.8s	2K

Table 5. Task creation costs

Table 5 lists the costs of the low level abstractions of ILI. Each heuristic was tested with two implementations. The first implementation uses task replication so that all tasks are pre-allocated on all computational units. Thus, task migration only employs state movement. The second implementation uses the UNIX command 'rsh' to startup remote tasks on demand and then transfers state. Clearly, task replication drastically reduces the amount of time required for this adaptation option.

Table 6 and Table 7 show the performance of ILI's two current adaptation heuristics in various dynamic situations. The results indicate that neither heuristic performs acceptably in all situations, which suggests that application-specific heuristics would further improve performance.

Event Rate for Instruments (in/out)					
Time	GCTM	S2G1	S2G2	S2G3	Annotate
			–	-	Data
5min	74(1x)	74	–	-	74
10min	148(2x)	74	74	-	110
15min	222(3x)	110	110	-	220
25min	296(4x)	110	110	-	220
30min	370(5x)	124	123	123	150

Table 6. Optimistically Greedy algorithm

Event Rate for Instruments (in/out)				
Time	GCTM	S2G	Annotate Data	VisWinds 2D
5min	74(1x)	74	74	74
10min	148(2x)	74	110	110
	GCTM	—	S2G & Annotate	VisWinds
15min	222(3x)	—	150	150
25min	296(4x)	—	270	270
	GCTM	S2G & Annotate & VisWinds		
30min	370(5x)	—	370	—

Table 7. Resource Conserving algorithm

7. Results and Contributions

This research addresses dynamic interactive high-performance applications. These applications have Quality of Service constraints that must be met due to their use of live input and their interactions with end users, which may change the applications' resource requirements, the resources available to them, the runtime behavior and possibly, their inputs and outputs. In addition, QoS constraints themselves may change with changing end user needs. The ILI infrastructure described and evaluated in this paper is able to accommodate such changes with little a priori knowledge of the applications or of their execution environments. This is achieved by use of a simple execution model for ILI via on-line monitoring and steering and with a method of load balancing that attempts to maintain event rates stated by QoS specifications.

The prototype implementation of ILI evaluated in this paper runs on a cluster of uni- and multiprocessor machines and has been used to manage the computational instruments of a large-scale interactive atmospheric modeling application. As constraints on the execution environment change, the ILI system detects constraint violations and corrects them using low cost decision heuristics and simple adaptation procedures, such as instrument cloning and task migration. Attempts are also made to reduce thrashing when switching between system configurations, by maintaining a configuration history.

8. Related Work

ILI tasks, their connections, and their runtime use may be modeled as a work flow system [10]. Extensive work in this area concerns organizational and execution models, typically focusing on the formal semantics of workflow specification, the flexible specification and generation of alternative worflows, and the application of certain analyses to workflow, such as the investigation of deadlines to workflow tasks. Furthermore, some work has been done on building distributed event-based workflow execution models similar to ours [6]. That work focuses on the formal semantics of modeling such event-based systems and designing them so that it is easy to express arbitrary sub-flows that can be instantiated into the existing workflow hierarchy during execution. In contrast, we provide a runtime infrastructure for support of the decision-making processes necessary when instantiating new sub-flows.

The primary goal in load balancing and migration systems [12] is to balance the load on computational units so as to increase utilization or prevent any one unit from slowing down the entire computation. This is useful if the constraints on the system are computation rather than interaction. In interactive systems, slowing down overall progress may actually be advantageous for purposes of debugging or visualization. Therefore, the state sought by the ILI framework is one in which a set of QoS constraints are met regardless of equilibrium with respect to computational load.

[11] describes a model and performance metrics for evaluating adaptive resource allocation (ARA) performed to satisfy application's real-time constraints. The model assumes a priori knowledge of the application's structure. This assumption is not made in our work.

[8] describes Schooner, an interconnection system designed to facilitate the construction of programs that requires access to heterogeneous software and hardware re-

sources. Applications built using Schooner are composed of pieces, called components, that communicate via specialized RPCs and are configured through a special configuration language which explicitly lists each possible configuration. Instead, ILI uses typed events as a method for implicitly and dynamically defining the possible ways in which components may be assembled.

9. Conclusions

The ILI infrastructure permits the specification of end-to-end constraints for clients that wish to interact with an application. It also provides the necessary ”glue” for the construction of distributed applications that span multiple LANs and even WANs. Such ILI-based 'interactivity systems' are capable of adapting to dynamic application behavior in order to maintain end-to-end constraints, in part by appropriate placement of intermediate processing nodes. By exposing its low-level adaptation primitives and its internal data, ILI also empowers the programmer to tailor adaptation heuristics to specific applications.

The two reconfiguration heuristics presented in this paper are examples of the manner in which quality of service levels may be guaranteed in dynamic environments. Our future work plans to examine additional heuristics and configurations with large numbers of tasks. to determine the low-level abstractions required for dealing effectively with premature task termination, and with hierarchies of ILI structures. We also plan to investigate the scalability of ILI for applications executing on platforms with large numbers of processors.

References

[1] R. A. Aydt. *Pablo Self-Defining Data Format*. Department of Computer Science, University of Illinois, 1304 West Springfield Avenue, Urbana, Illinois 61801, May 1993.

[2] G. Eisenhauer. Portable self-describing binary data streams. Technical Report GIT-CC-94-45, College of Computing, Georgia Institute of Technology, 1994.

[3] G. Eisenhauer, B. Schroeder, K. Schwan, V. Martin, and J. Vetter. High performance communication in distributed laboratories. *To appear in Journal of Parallel Computing*, January 1998.

[4] G. Eisenhauer, K. S. Weiming Gu, and N. Mallavarupu. Falcon - toward interactive parallel programs: the on-line steering of a molecular dynamics application. In *Proceedings of High-Performance Distributed Computing 3*. IEEE, August 1994.

[5] M. Forum. MPI: A message passing interface standard. Technical report, University of Tennessee, 1995.

[6] A. Geppert and D. Tombros. Event-based distributed workflow execution with eve. Technical Report 96.05, University of Zurich, Department of Computer Science, University of Zurich, September 1997.

[7] J. K. Hollingsworth and B. P. Miller. Dynamic control of performance monitoring on large scale parallel systems. In *Proceedings of International Conference on Supercomputing*, Tokyo, July 1993.

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

[9] T. Kindler, K. Schwan, D. Silver, M. Trauner, and F. Alyea. A parallel spectral model for atmospheric transport processes. *Concurrency: Practice and Experience*, 8(9):639 – 666, November 1996.

[10] M. Kradolfer and A. Geppert. Modeling concepts for workflow specification. Technical Report 5, University of Zurich, May 1997.

[11] D. Rosu, K. Schwan, S. Yalamanchili, and R. Jha. On Adaptive Resource Allocation for Complex Real-Time Applications. *18th IEEE Real-Time Systems Symposium*, Dec. 1997.

[12] A. Sohn, R. Biswas, and H. D. Simon. A dynamic load balancing framework for unstructured adaptive computations on distributed-memory multiprocessors. *Symposium on Parallel Algorithms and Architectures*, 1996.

[13] J. Vetter and K. Schwan. High performance computational steering of physical simulations. *IEEE Proceedings of the Internatl Parallel Processing Symposium*, pages 128–132, 1997.