An Object-Based Infrastructure for Program Monitoring and Steering

Greg Eisenhauer and Karsten Schwan

College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332

Abstract

Program monitoring and steering systems can provide invaluable insight into the behavior of complex parallel and distributed applications. But the traditional event-stream-based approach to program monitoring does not scale well with increasing complexity. This paper introduces the Mirror Object Model, a new approach for program monitoring and steering systems. This approach provides a higher-level object-based abstraction that links the producer and the consumer of data and provides a seamless model which integrates monitoring and steering computation. We also introduce the Mirror Object Steering System (MOSS), an implementation of the Mirror Object Model based on CORBA-style objects. This paper demonstrates the advantages of MOSS over traditional event-stream-based monitoring systems in handling complex situations. Additionally, we show that the additional functionality of MOSS can be achieved without significant performance penalty.

1 Introduction

As applications have grown more complex, the need for their developers to gain insights into their run-time behavior has increased. These insights are now critical for creating, understanding, debugging and tuning those programs. Furthermore, additional productivity gains can be attained by allowing users to control program behavior during execution through program steering.

The development of application-level program monitoring and steering tools has done much to promote the separation of the application and the complex program monitoring and steering interfaces that support extensive user interaction. These tools have made it possible to add interactivity to high-performance applications without extensive application restructuring. They enable the construction of both generic and application-specific clients that attach to, extract information from, and interact with the core application. We call the set of clients which interface with an application an *interactivity system*.

This paper introduces the Mirror Object Model, a new approach for program monitoring and steering systems. This approach provides a higher-level object-based abstraction that links the producer and the consumer of data and provides a seamless model that integrates monitoring and steering computation. We also introduce the Mirror Object Steering System (MOSS), an implementation of the Mirror Object Model based on CORBA-style objects. The paper demonstrates the advantages of MOSS over traditional event-stream-based monitoring systems in handling situations that arise in complex interactivity systems. Additionally, we show that the added functionality of MOSS can be achieved without incurring significant performance penalty.

To appear in the 2nd SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT'98), August 1998

2 Influences and Goals

The design of the Mirror Object Model was influenced by the Distributed Laboratories Project[13] at Georgia Tech and its goals of supporting complex and collaborative interaction with long-running distributed and parallel applications. As an illustrative example of the type of application targeted by Distributed Labs, consider a long-running scientific simulation, such as a global climate model described in [9]. This atmospheric model achieves high simulation speeds by distributing the simulation of different atmospheric layers across processors on an SMP or network of workstations. In a Distributed Laboratory, the progress of this application can be monitored and controlled by multiple scientists at spatially distributed locations. Monitoring is used to capture certain application quantities, such as global wind data, concentration of certain airborne chemicals, and localized climate data in critical areas. The scientists use this information to evaluate the model and adjust it if necessary.

An online program monitoring and steering system seems natural for this monitoring and control task, but existing monitoring systems cannot easily scale to accommodate the large volume of scientific data produced by a parallel climate model. Prior research in program monitoring has developed mechanisms for controlling the overall flow of data. But some data reduction schemes, such as on-off switches on monitoring sensors, do not lend themselves well to situations where there is more than one end-user[7, 17, 10]. In any case, such data suppression mechanisms are of little use if the consumers really need all of the data being generated.

Additionally, the quantity and form of data potentially required by atmospheric displays is such that the calculations necessary to transform the data for display may themselves need to be done in parallel. In the case of atmospheric modeling data, each atmospheric level must be converted from spectral to grid form, an operation that is embarrassingly parallel. Yet many monitoring systems are built on an event-stream abstraction. That is, they treat the application and monitoring clients as processes linked by a stream of data records. Some event-stream-based systems can direct or filter monitoring data records by type, but they generally do not directly support content-based event routing[7, 17]. This limitation simplifies the event handling infrastructure, but such systems cannot handle the selective data distribution necessary to do data-parallel monitoring distribution. In addition, some systems draw a sharp distinction between the application and monitoring clients, making it difficult to build multi-level data reduction schemes[10, 11].

2.1 Goals for a New Model

Based on the difficulties described above, we established goals for a new model for program monitoring and steering systems:

- scale gracefully with increasing complexity in both the application and the monitoring system;
- provide for automatic control of monitoring data collection in the presence of many simultaneous clients;
- permit self-application for multi-level processing and control; and
- provide a natural model of steering.

The first three goals target the problems in creating the complex, multi-level and data-parallel monitoring systems which were exposed above. The need for self-applicability is relatively straightforward when one considers multi-level monitoring data reduction schemes, but we believe that it is also important for steering. While the need for multi-level steering at the application level is unexplored, a more obvious necessity derives from the need to perform load-balancing and other control in complex monitoring systems. As the interactivity system becomes more complex, its performance becomes increasingly important to overall application behavior and resource usage, and it becomes a useful target for monitoring and steering in order to control its behavior.

The last goal derives from a more generic difficulty, that there is no obvious counterpart to monitoring through which program steering can be accomplished. Unlike monitoring, which is implicitly synchronous, steering involves modifying the state of the running application by an external agent. In general, such modifications cannot be safely performed without some application-level synchronization. Several approaches to steering synchronization have been proposed. Progress[17] performs steering actions only at specific execution points. Autopilot[16] allows steering updates to be enabled and disabled on an object-by-object basis. But no system allows the natural use of synchronization code which may already be a part of some parallel and distributed applications.

3 A New Model for Monitoring and Steering

In order to meet the goals enumerated above, we propose a new model for program monitoring and steering. For clarity we first describe the model without reference to a specific implementation.

Our approach exploits the idea that the interactivity system *augments* the original application. That is, the monitoring and steering system provides functional behavior, access

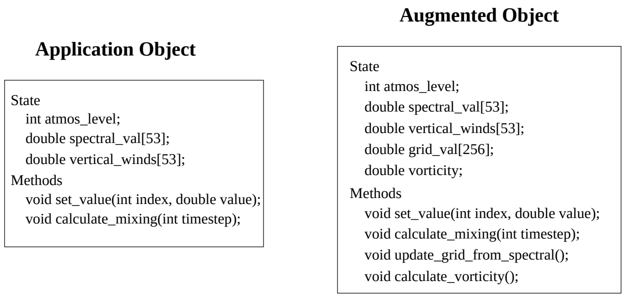

Figure 1: An application object augmented with interactivity methods and state.

or interfaces that were not included in the core application, but are valuable to its users. In order to accomplish this augmentation, the Mirror Object Model treats application-level entities as objects with associated methods and state. The code required to perform computations within the interactivity system is viewed as additional methods associated with these objects, and any supplemental state variables required by these calculations are additional state variables associated with the objects. This model, depicted in Figure 1, provides a conceptual basis for the relationship between application entities and the interactivity system.

However, requiring the interactivity state and methods to be physically added to objects in the application would not achieve the goal of creating a system for distributed monitoring and steering. In order to support the conceptual model above in a distributed environment, we do not locate the interactivity methods and state in the application, but rather place them in distributed interactivity clients in "mirror objects." These mirror objects are analogues to the objects in the application program, sharing their state (via monitoring) and implementing their methods (via remote object invocations), as well as having additional state and methods for interactivity computations. These additional methods and state provide a mechanism through which the interactivity system can perform monitoring data reduction or accomplish the computation necessary for algorithmic steering. The act of steering is performed through the application object's original methods via remote object invocation. Figure 2 shows an application object and its mirror object in a monitoring system client.[1]

Application objects may be mirrored at multiple points in the interactivity system, thereby providing support for multiple observers and operators. Mirror objects themselves may be also be further mirrored at other points in the system. This capability is useful in the progressive refinement and reduction of monitoring data, and also in the creation of meta-level systems which monitor and steer the interactivity system itself. Finally, the interactivity system can also host new objects which do not directly correspond to any application object. These new objects may themselves be mirrored at other points in the interactivity system and further enrich the computational model.

The Mirror Object Model of program monitoring and steering employs object-oriented principles and is particularly suitable for adding interactivity to both traditional

[1] For simplicity, we describe the basic model mirroring the entire state of the object uniformly. However, this figure anticipates the ability to specify different update characteristics on a per-attribute basis.

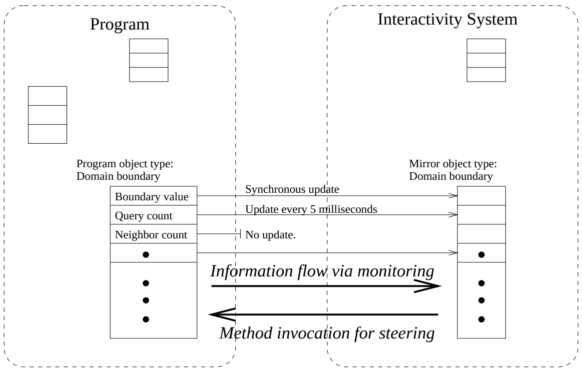

Figure 2: An object in the application and its mirror object in the interactivity system.

```
/* boundary.idl
**
** An object representation of the particle
** domain boundaries.
*/
interface boundary {
  attribute float boundary;
  attribute long index;

  float get_val();
  void put_val (in float value);
};

interface domain {
  attribute long mol_count;
  attribute long index[3];
};
```

Figure 3: An example of an object specification in IDL.

and object-based high performance applications. In object-based applications, the approach of making application-level object methods available to the interactivity system means that steering is essentially "free." That is, interactive application steering is made available without requiring the application programmer to write specific code to accomplish it because the existing object methods are employed. Since the programmer of a distributed object-oriented application has already spent considerable time designing his objects' methods and managing the synchronization necessary to maintain information consistency, leveraging this effort for use by the interactivity system is an important consideration.

4 Implementation

The abstract Mirror Object Model described in Section 3 can be realized in virtually any object system. This section will discuss implementation considerations and describe the choices made in our implementation of the model, the Mirror Object Steering System (MOSS). Because the clients in the interactivity system are more likely to be developed from scratch, that system is discussed separately from the application, where it is more likely to be necessary to deal with an existing program.

4.1 In the Application

The Mirror Object Model views application components as objects with associated state and methods. This is obviously a semantic match if the application is indeed written in an object-oriented style. But even for applications written in procedural languages, the mirror object approach is still valid. Any simple procedural program will require modification in order to support application-level monitoring and steering. Systems such as Falcon[6] and Progress[18] require the insertion of code snippets into the application at appropriate points. A similar approach which is more agreeable with the object-based nature of the interactivity system is to "objectify" small portions of the application to enable them for steering. The term "objectify" means to identify the components of application state which are of interest for monitoring and steering and "wrap" them with an object-like interface. Given that such an application would almost certainly require modification to allow safe steering, incorporating an object-like abstraction does not present significant additional difficulties.

The implementation of object monitoring and remote invocation depends upon the nature of the object system employed in the application. Many object systems provide for remote object invocation. Of those that do not, some will export sufficient information to permit a remote invocation system to be constructed without modifying the program. As a last resort, the program may be modified to identify and register object classes and methods with a custom remote invocation system. The implementation of monitoring may be performed in a similar fashion.

The approach taken in MOSS is to use a CORBA-style object system. Since remote invocations are part of basic CORBA functionality, this provides us with our steering mechanism. However, if such a system was not available, one could resort to program annotation (such as Falcon[7] and Progress[18]), code transformation (as in Pablo[14]) or, if sufficient information was available in the program, dynamic instrumentation techniques (such as those used in Paradyn[10]) to enable monitoring.

Our MOSS implementation follows the style common to CORBA Object Resource Broker implementations. Object interface declarations, such as the example given in Figure 3 are written in IDL. These IDL declarations are then processed by a stub generator to generate header files and program stub routines that are suitable for use in the target language. These stubs provide support for local and remote method invocation and access to the class variables. The application programmer then supplies the implementation for the actual methods. For example, the IDL stub generator creates C code and headers and requires the object method implementations to be written in C. These stubs, together with the method implementations provided by the programmer, completely implement the object specified by the IDL interface. These objects can then be used to create new applications, or they can be integrated into existing applications.

4.1.1 Instrumentation

Function. Instrumentation, in the context of the Mirror Object Model, is the mechanism through which changes in application object state are detected and transmitted to the mirror objects in the interactivity system. The model described in Section 3 requires that changes in object state be mirrored in the interactivity system and that those changes cause `Update()` methods to be called in the mirror objects. This specification leaves room for variations in implemen-

```
void
impl_boundary__set_index(boundary o, CORBA_long value,
                         CORBA_Environment *ev)
{
   _IDLtoC_boundary_attrib_struct *state;
   static atom_t otl_self_atom = 0;
   static atom_t otl_chan_handle_atom = 0;
   ECSourceHandle event_handle = NULL;
   if (otl_self_atom == 0) {
        otl_self_atom = attr_atom_from_string("OTL:OBJ_SELF");
        otl_chan_handle_atom =
            attr_atom_from_string("OTL:MON_CHANNEL");
   }
   query_attr(o->obj_name, otl_self_atom,
              NULL, (void**)&state);
   query_attr(o->obj_name, otl_chan_handle_atom,
              NULL, (void**)&event_handle);

    state->index = value;
    if (EChas_sinks(event_handle)) {
        boundary_submit_state(event_handle, state);
    }
}
```

Figure 4: The code generated for the set procedure for the attribute `index` in the example class `boundary`.

tation. The approach taken in our initial implementation of MOSS reflects each change in object state immediately to the mirror objects, but some other approaches are worth noting. For example, Figure 2 depicts different update specifications applied to each attribute in the mirrored object. Attribute update specifications might explore the semantic variations that were done with monitoring sensors in [6], that is, variable-rate tracing sensors and sampling sensors. The analogous techniques in MOSS would involve arranging instrumentation such that object state is reported less frequently than at every change (such as every Nth change) or that data values be reported at regular time intervals regardless of change. Because these techniques were evaluated in detail in [6], they are not considered further here. However they are very appropriate for MOSS and may be employed in the future to further reduce monitoring overheads.

Implementation. In CORBA, access to object attributes occurs only through get/set routines that are generated by the compiler. This simplified the implementation of MOSS because attribute instrumentation could be accomplished simply by modifying the IDL compiler to generate instrumentation in the attribute set routines. Figure 4 shows the code that is generated for the set procedure for the attribute `index` in the class `boundary` shown in Figure 3. Most of the routine is the implementation of the attribute set functionality and relates to the functionality of OTL, the Object Transport Layer infrastructure that provides run-time support for our CORBA system.[2] The final `if` statement and the subroutine call it protects represent the embedded instrumentation.

MOSS uses event channels to transport monitoring information to the mirror objects in the interactivity system. Event channels are a publish-subscribe communication mechanism similar to the artifact of the same name described in the CORBA Common Object Services specification[1]. However, unlike object-level implementations of the CORBA Event Services specification, our event channels do not rely upon a central object for event distribution. Instead, event communication from source to sink occurs directly, regardless of the number or location of sources and sinks. If there are no sinks for a particular channel, no network traffic occurs. Event channels and state marshaling are implemented using DataExchange and PBIO[4], communications infrastructure packages which were developed for the Distributed Laboratory environment. Event channels were developed together with MOSS, but they are in no way dependent upon MOSS or the object system and can be used independently.

The `EChas_sinks()` call in the `if` conditional checks to see if there are any subscribers to this object's event channel that will receive events. If there are none, this check bypasses the remainder of the instrumentation and no events are generated.[3] Because MOSS uses one event channel per monitored application object, this allows per-object configurability of monitoring. This is a much finer level of control than has been possible in previous systems and is a valuable mechanism for holding down monitoring overhead.

Mirror objects in the interactivity system subscribe to the event channel of the object they mirror, and the semantics of the event channel ensure that events are routed only to interactivity clients that are interested in receiving them. The channel also naturally takes care of pairing events with the appropriate object when the event arrives at the client, as well as suppressing the transmission of monitoring data when there are no listeners. The suppression of unwanted monitoring events is vital in helping the monitoring system handle the volume of potential monitoring information. Reducing the flow of unnecessary events is critical to the scalability of any monitoring system and is often handled by on-off switches associated with application sensors, a technique that becomes harder to manage when there is more than one event consumer. That event suppression can be done on an object-by-object basis and that it operates automatically, even in the presence of multiple consumers, is a strong advantage for the mirror object system.

An equally important benefit of the implicit routing provided by event channels is that it solves the scalability problems raised in the example in Section 2. With event channels managing data flow between the application objects and the mirror objects in the interactivity system, the decomposition of application and interactivity processing no longer complicates the system. For example, events from atmospheric layer N are routed only to the client which is to process them and are automatically matched with recorded state in the mirror object. This occurs regardless of the processor-wise decomposition of the atmospheric model or of the data reduction computations. This behavior is examined further in Section 5.

4.1.2 Steering

Function. In the Mirror Object Model, program steering is accomplished by remote invocation to the application objects. As described in Section 3, this is a natural choice if the application is written in an object-oriented style and means that steering is available without requiring the application programmer to write specific code to accomplish it. It also allows the Mirror Object Model to leverage any synchronization code already written by the application developer to arbitrate access to application objects. If the application

[2]OTL is described further in [3].

[3]The `EChas_sinks()` check is not strictly necessary because the event channel would not have transmitted the event if there were no subscribed sinks. However, performing the check allows us to avoid the overhead of encoding the object state for transmission. This encoding is done before submission to the event channel.

```
extern CORBA_float
  boundary__get_boundary(boundary o, CORBA_Environment *ev);
extern void
  boundary__set_boundary(boundary o, CORBA_float v,
                         CORBA_Environment *ev);
extern dimension
  boundary__get_dim(boundary o, CORBA_Environment *ev);
extern void
  boundary__set_dim(boundary o, dimension v,
                    CORBA_Environment *ev);
extern CORBA_long
  boundary__get_index(boundary o, CORBA_Environment *ev);
extern void
  boundary__set_index(boundary o, CORBA_long v,
                      CORBA_Environment *ev);
extern CORBA_float
  boundary_get_val(boundary object, CORBA_Environment *ev);
extern CORBA_void
  boundary_put_val(boundary object, CORBA_float  value,
                   CORBA_Environment *ev);
```

Figure 5: The declarations generated for the IDL specification in Figure 3.

is not object-based, or if its existing synchronization mechanisms will not permit safe steering, then the application will have to be objectified and/or augmented with additional synchronization.

Implementation. Our initial CORBA-based implementation of the Mirror Object Model, MOSS, is particularly suitable for objectifying certain parts of non-object-based applications to make them available for monitoring and steering, but it could also be used directly in CORBA-based applications or used to develop such applications from scratch. As described above, the IDL compiler generates stub routines for object methods based on the IDL specification. These stubs provide mechanisms through which application programs can invoke object methods, as well as support for remote invocation. The precise nature of the stub routines is more a function of the supporting object system than of the Mirror Object Model. However, the set of routines available to the programmer and those which must be supplied by the object implementor are discussed as these are of direct interest to the programmer in adding monitoring and steering to an existing program.

Figure 5 shows the declarations generated by the IDL compiler for the example IDL specification given in Figure 3. The majority of these routines are simple access routines for the object attributes, but `boundary_get_val()` and `boundary_put_val()` are stub routines which give access to the object methods. In addition to those stub routines, the IDL compiler also generates declarations for the method implementations, given below:

```
extern CORBA_void
impl_boundary_put_val(boundary object, CORBA_float value,
                      CORBA_Environment *ev);
extern CORBA_float
impl_boundary_get_val(boundary object, CORBA_Environment *ev);
```

The bodies of these routines must be provided by the class implementor. In the case of the example object, the `boundary_put_val()` method is designed to be used to asynchronously modify the position of the "boundary," an element of a geometric decomposition in a physical simulation. The position of this boundary can be changed at any time during the simulation, but processors must be guaranteed a consistent view of the configuration. Therefore, the body of the `impl_boundary_put_val()` acquires a global lock before using the `boundary_set_boundary_value()` routine to change the state of the object. This code is shown in Figure 6.

```
CORBA_void
impl_boundary_put_val(boundary o, CORBA_float value,
                      CORBA_Environment *ev)
{
    thr_mutex_lock(configuration_lock);
    boundary_set_boundary(value);
    thr_mutex_unlock(configuration_lock);
}
```

Figure 6: The implementation of `impl_boundary_put_val()`.

Application-level synchronization is an important and difficult issue in program steering. The Mirror Object Model does not impose any particular model for synchronization. For example, some object systems, such as Java, provide models of synchronization that are tightly coupled with the object system itself. In particular, Java allows some objects to be designated as monitors, with the result that only one invocation at a time is allowed to be active within them. In other systems, such as in our initial CORBA-based MOSS implementation, synchronization is entirely up to the application programmer. This flexibility allows MOSS to leverage already-programmed synchronization code where available, as well as enabling it to emulate the synchronization styles of Progress[18] and Autopilot[16] with application-level code.

4.2 In the Interactivity System

The issues involved in creating clients in the interactivity system are very similar to those described in Section 4.1. In particular, because the Mirror Object Model is meant to be applied to itself to enable meta-level monitoring and steering, mirror objects in the interactivity system have many of the same features and responsibilities as objects in the application. Implementations supporting the general Mirror Object Model could support clients written in any number of OO languages, from SmallTalk to Java, but in MOSS the interactivity system programming environment is derived from the same CORBA-style object system used at the application level. To specify the additional methods and state of the mirror objects, IDL is extended with a "mirrors" keyword that is used in the IDL interface header declaration and operates much like an inheritance specification. Where in normal IDL the specification:

```
interface ibound : boundary {
       ....  additional attributes and methods ...
};
```

declares that the class `ibound` was derived from the class `boundary`, the mirror specification to declare a class `ibound` which mirrors objects of class `boundary` appears as:

```
interface ibound mirrors boundary {
       ....  additional attributes and methods ...
};
```

The principal difference in compiler behavior for the latter case is that the compiler generates remote object invocation code for the mirrored methods. Additionally, if one of the methods of the new class is named "update," the compiler generates code to ensure that this method is called when new object state arrives via monitoring. This automatic method call upon update provides a triggering mechanism for interactivity system calculation, which may include monitoring data reduction or algorithmic steering.

Another significant optimization opportunity provided by the Mirror Object Model relates to the potential for relocating interactivity computations. Because the Mirror Object Model conceptualizes those computations as methods associated with the application object, it is possible to analyze those methods and potentially relocate them in order to improve performance, reduce latency or balance loads in the interactivity system. While the general case of relocating a method from an interactivity client to another client or into the application may be prohibitively complex, there are some obviously simple cases where it is possible to reap significant benefits from this ability. For instance, a monitoring client may not operate on every piece of updated data it receives, but instead apply a simple threshold function and operate only if the data exceeds the threshold value. Moving this threshold operation to the source of data eliminates unnecessary data transmission. Similarly, simple algorithmic steering operations may be moved from the interactivity system directly into the application. Because latency can be a significant limitation on the applicability of algorithmic steering, migrating externally-specified steering operations directly into the application can open new domains for program steering. For example, [12] describes mutex locks that adapt themselves to the most efficient mode of operation based on their usage pattern. Normally this sort of adaptation would be beyond the ability of external steering because of the response time required for adaption, but steering actions within the application should be capable of such rapid response times. This optimization is discussed further in Section 5.2.

5 Performance

This section evaluates MOSS in terms of the goals described in Section 2.1. In particular, the following evaluations are presented:

- **Sensor production microbenchmark** – To support MOSS' claim to be a monitoring and steering system suitable for high-performance applications, We must show that support for higher-level object abstractions does not overly impact the basic functions of a monitoring and steering system. One such measure is the amount of overhead involved in monitoring. This experiment compares the time required to generate monitoring events of various sizes in each of three systems: MOSS, Audobon (a variant of Falcon, a previous Georgia Tech monitoring system) and Autopilot from the University of Illinois.
- **End-to-end steering benchmark** – a second basic measure of the function of a monitoring and steering system is the time required to complete a round-trip steering action triggered by a monitoring event but with the steering action initiated by a process on another machine. Again, the goal is to show that MOSS does not impose substantial additional overhead. To demonstrate this we perform similar steering tasks with MOSS and Autopilot and compare their performance.

  Additionally, to show the importance of steering action migration, the results of the basic end-to-end two-machine steering benchmark are compared to the time required to perform a steering action when that action has migrated from an external client into the application itself. Steering action relocation, as described in Section 4.2, does not occur automatically in MOSS, but the MOSS framework is designed to support such optimizations. This experiment demonstrates the value of this particular optimization.
- **Data parallel data reduction** – another important claim of the Mirror Object Model is that it is effective in scaling the interactivity system because it allows automatic routing of monitoring information to the proper data reduction point. This contrasts with more traditional event-stream based monitoring systems which must broadcast events. Routing updates only to interested clients becomes important in several situations. One of these is when the quantity of monitoring data requires parallelizing the data reduction system in order to meet throughput requirements. We show that the Mirror Object Model and MOSS can significantly reduce network throughput requirements.
- **Automatic data suppression** – an equally important capability for scaling in a monitoring and steering system is the suppression of unwanted monitoring events. The schemes employed by many systems do not handle multiple clients well because they involve explicit switches to turn monitoring sensors on or off. This experiment demonstrates the ease with which the Mirror Object Model handles complex situations. The experimental setup involves a data source exporting data objects which represent a grid of values which is updated regularly. Element-by-element data is available as well as aggregate values which represent the average value of some region of the grid. The clients of this data source are interested in receiving element-by-element data for a portion of the grid around their point-of-interest and average values for the remainder of the grid. Each client's point-of-interest can change over time, and there may be more than client receiving updates at the same time.

5.1 Sensor Production Microbenchmark

One possible criticism of the Mirror Object Model is that support for the extended functionality it offers will increase the costs of basic monitoring and steering operations. In order to address this concern, this section examines the computational costs of a basic monitoring operation in MOSS as well as two other systems which involve monitoring. The first of these external systems is Audobon, a variant of the Falcon[7] monitoring system developed at Georgia Tech. Audobon was chosen because it is a basic event-stream-based monitoring system. The second external system chosen for evaluation is Autopilot[16], a program monitoring and steering system recently developed at the University of Illinois. Like MOSS, Autopilot is object-based and implements per-object data routing facilities.

Because the amount of overhead imposed by network transmission is both potentially large and mostly outside the control of the monitoring system, the experimental setup separates the network from consideration and concentrates on computational costs which precede the network `write()` call. In each of the three systems, this is accomplished by interposing a null network send routine into the monitoring system infrastructure after allowing the monitoring system to establish a normal connection to an external client. In the case of Audobon and MOSS, both of which use PBIO[2, 4] for network transmission, this interposition is accomplished

Data Size	MOSS	Audobon	Autopilot	Transmit Time
12	14.4 μs	26.3 μs	24.9 μs	4.29 μs
100	14.4	27.8	25.6	5.34
1024	14.4	33.5	38.0	14.1
10240	14.4	88.6	157.	182.
102400	14.4	641.	1260.	1510.

Table 1: Non-network computational costs for sensors of various sizes in MOSS, Audobon and Autopilot. Network transmission times included for reference.

by using `set_interface_IOfile()` to modify the basic functions through which PBIO accesses the network and supplying a `write_func` which returned a success code. In the case of Autopilot, which uses Nexus[5] for network transmission, internal Nexus data structures were traversed and the protocols `send_rsr` function pointer was changed. Instead of a pointer to `tcp_send_rsr()`, this pointer was redirected to a function which deallocated the buffer and returned. (Buffer deallocation would normally occur at the completion of the network write in Nexus.)

Because the Mirror Object Model targets include large scientific simulations which may produce very large events, sensor costs were measured for a variety of sensor sizes. This experiment was run on a Sun Sparc Ultra 1 workstation and the results are shown in Table 1. Also included for reference is the raw TCP/IP transmission time for messages of the given size between UltraSparc1's connected by 100Mbps ethernet.[4]

The striking aspect of the results in Table 1 is that while the non-network computational costs in MOSS do not grow with sensor size, those of Audobon and Autopilot do. This dramatic difference in behavior is not *per se* a reflection of the broad design characteristics in the systems, but can instead be traced to more subtle differences in semantics and operation.

In Autopilot, the principal performance impact comes from two copy operations that are performed on the data before it reaches the network `write()` call. Both of these copies are the result of the choice of Nexus as the transport mechanism. The first copy is the result of reliance on the Nexus put/get operations for filling the buffer with primitive data types. Nexus does not impose a "network format" for data, so the put operation on a primitive data type does not actually transform or byte-swap it, but it does result in a copy of the data into a send buffer. The second copy operation is internal to Nexus and is a result of a Nexus design decision. Rather than allowing the application thread to write to the network directly, the Nexus `tcp_send_rsr()` call instead queues the data for later sending by a network handler thread. However, since the application is allowed continue executing and the network write may not happen immediately, Nexus must first preserve the data to be sent by moving it to another buffer and incurring a second copy operation. This "out-of-line" network send operation has the advantage that the application thread is not blocked for network operations, but in this case has the consequence of imposing additional copy overhead. Neither of the two copy operations which impact Autopilot sensor performance is necessary in the sense that it is unavoidable in terms of Autopilot semantics, and further optimization of the Autopilot transport system might significantly reduce this overhead.

[4]Transmission time is derived from measured TCP/IP bandwidth at the given message size on the test platform. Bandwidth measurements were made with NetPerf[8].

In Audobon, the additional overhead as data size increases can also be traced to a copy operation. Audobon, like Falcon, has sensors which accept data as subroutine parameters. Additionally, the functional requirement for an Audobon sensor requires that it add certain header information to the data. However, its sensors were designed to accommodate the limitations of an earlier version of PBIO which required at least the base data block to be a "C"-structure-style contiguous block. A copy operation was necessary to achieve contiguity. However, the semantics of Audobon also do not intrinsically require a copy operation. It is built on top of PBIO, which is now designed to write application-level data in-place, so a reimplementation of Audobon and its sensor mechanism could also potentially eliminate data-dependent growth in sensor costs.

In contrast to the two systems above, MOSS does not normally incur additional non-transmission costs as such things as array sizes grow in monitored data. This is due to careful design and interaction in the data marshaling mechanism (PBIO) event system which routes and transports. First, the event system accepts data payload for an event as either a single contiguous buffer, or a list of buffers and sizes (i.e., a buffer vector). The latter dovetails with PBIO's ability to generate an encoding of a complex data structure and represent that encoding as a buffer vector. In particular, PBIO's `encode_IOcontext_to_vector()` call is used to marshal the sensor data. This encoding requires the creation of a small amount of data describing the structure to be encoded. Additionally, any substructures containing pointers (to strings or dynamic arrays) must be copied in order that their pointers can be rewritten into offsets in the message. However, the majority of the data, particularly large arrays of atomic data types, is left at its application address, with only its address and size appearing in the buffer vector. Since the event channel operates with the semantics that the complete processing of the event data payload before the event submission returns, no data copying is necessary.

This experiment shows that MOSS is more efficient than Autopilot, a monitoring and steering system which is also object-based, and even more efficient than Audobon, a much less featured basic monitoring system. It also underscores the importance of such basic techniques as minimizing copying in data transfers.

We also note that minimizing transmission overhead is only one aspect affecting a monitoring system's ability to handle large amounts of data. Other potentially useful techniques include the buffering of updates, periodic sampling instead of push-style update generation, and the local reduction of data through averaging and other means. These facilities were explored in Falcon[6] and are available in Autopilot. It is possible to employ such techniques to the object state update mechanism in MOSS as well. Additionally, one could reduce overheads in MOSS by only transmitting changed attributes when an update occurs. However, as prior research has already explored these techniques their application to MOSS is left for later development.

5.2 End-to-end Steering Benchmark

The basic steering functionality in MOSS will be evaluated by comparing its performance with that of Autopilot. While Autopilot is also an object-based monitoring and steering system, these systems do differ in the amount of overhead inherent in their approaches. Autopilot's actuators have just a single method, processActuatorData, that can be triggered by remote clients. Parameters to that method consist only of a single array of atomic data types. The run time system needed to support this functionality is not as complex as the MOSS run-time system, which must be capable of directly invoking any application-level method.

To evaluate this cost, similar steering situations were created in Autopilot and MOSS. Each involved an application-level sensor triggering a steering action from a remote client. The processing on the client is minimized, consisting of only initiating the steering action, passing as a parameter the data it has received from the sensor. The application-level steering action consist only of copying the parameter data into an application-level data structure. The experiment was first run between two Sun UltraSPARC 170's connected with 100Mbps ethernet. The results are shown in Table 2. The times reported are elapsed real-time between sensor initiation and the completion of the steering action.

The results in Table 2 show that the steering response times for MOSS and Autopilot are comparable. Autopilot is 30% faster than MOSS for the smallest data set measured. But for large data exchanges, more efficient data handling gives MOSS the advantage. The use of a more efficient distributed object system than OTL could further reduce the MOSS round-trip steering time. However, we believe that the additional flexibility and functionality offered to MOSS clients outweighs the performance differences in the current implementation.

A more ambitious optimization also has great potential for improving the performance of MOSS as was mentioned in Section 4.2. Other systems, such as Autopilot and Progress, make a clear distinction between the sensors which produce monitoring data and the actuators which can be used to steer the program. In the Mirror Object Model, sensors and actuators are combined in the form of the mirrored application-level object. This provides a semantic link, missing in other systems, that can be exploited for automatic optimization. In particular, this link gives MOSS the potential to migrate external steering computations into the application. As mentioned in Section 4.2, avoiding the round-trip across the network can potentially reduce steering latency dramatically and open new domains of application to steering. General application steering migration is almost certainly impractically complex, but in certain simple cases it is relatively easy. For example, consider the following Update() method from a steering client. The method is represented in C++ and the variable request_rate is an object state variable mirrored from the application object. The method Replicate is also mirrored from from the application object and represents a call that steers the application object.

Table 2: Elapsed real-time between sensor initiation and steering completion.

Data Size	MOSS	Autopilot
12	18.6 msec	13.4 msec
100	19.8	14.3
1024	22.2	18.1
10240	45.6	43.5
102400	266.	297.

```
SteeringClient::Update()
{
    if (request_rate > 100) {
        Replicate();
    }
}
```

This is a very simple, but useful type of steering computation. It references nothing outside of the object itself, but it does accomplish a basic steering goal. It monitors an important internal aspect of object operation and reconfigures the object when a threshold is exceeded. This type of steering computation would be an ideal candidate for migration into the application. Because of its simplicity, a compiler could easily determine that this method had no external side-effects and depended only on mirrored data. If the migration could be accomplished, steering latency could be dramatically reduced and network traffic eliminated. As an example of the gains possible, Table 3 shows the steering latency numbers of Table 2 with an added column that represents the time required if the steering action in the earlier example was migrated into the application. The numbers shown here were **not** produced through automatic steering migration, but are meant to show the benefits that can be gained through that optimization.

Table 3: Elapsed real-time between sensor initiation and steering completion with steering action migration.

Data Size	MOSS	Steering Migration
12	19.8 msec	0.304 msec
100	20.7	0.363
1024	22.2	0.538
10240	45.6	1.91
102400	266.	16.3

The results in Table 3 show clear and dramatic benefit from migrating steering computations. The results above were generated by manually linking the steering code into the application and causing it to be called in response to a sensor event. While MOSS is not yet capable of performing this important optimization automatically, this potential optimization is still a valuable contribution of MOSS and will be an active area of future work. The most significant obstacle in achieving these gains automatically is the practical difficulty of moving the Update() method code into the application. One promising approach is the use of dynamic code generation, such as is employed in MDL in the Paradyn project[10]. As described and measured in [4], PBIO is already capable of using dynamic code generation to create customized data conversion routines. These generated routines must be able to access and store data elements, convert elements between basic types and call subroutines to convert complex subtypes. Generating a simple Update() routine like the one shown above merely requires adding expressions and simple control structures to PBIO's repertoire.

5.3 Data Parallel Data Reduction

In order to demonstrate the abilities of MOSS with respect to data parallel data reduction, we will describe an application situation and detail the manner in which it could be implemented in MOSS and with a more traditional event-stream-based monitoring and steering system. Then we will calculate the network bandwidth required by each solution.

For the application, we once again consider the atmospheric model introduced earlier. This model performs most of its computations in the spectral domain, with atmospheric species concentrations represented by superimposed waves with different weights. Yet the most meaningful representation for display of this data requires the spectral representation be converted to a grid-based representation. This requires evaluating the spectral representation at each grid point and is a computationally costly conversion. If the atmospheric model is executing rapidly, it can produce spectral data more quickly than the display processor can convert it to grid form. In this situation, it is natural to want to parallelize the spectral-to-grid conversion in order to achieve the required conversion speed. In particular, because species data for each atmospheric level can be converted independently, a data-parallel approach leads to an embarrassingly parallel solution.

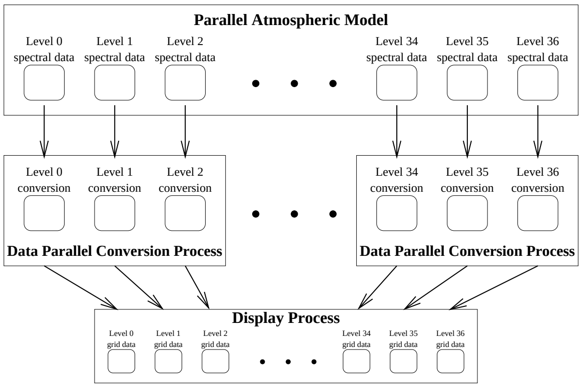

Figure 7: Logical data flow in data parallel monitoring reduction.

Figure 7 shows data locations and message flow for a data parallel decomposition of the spectral-to-grid conversion. The parallel atmospheric model is depicted as a single process as it might be on a large shared memory multiprocessor. If it were decomposed across distributed memory processing elements, the data flow between the application and the conversion processes would not change. Each arrow represents a transmission of data associated with a single atmospheric level, with the upper set of arrows representing spectrally encoded data and the lower set representing data in grid form. In this figure, we have arbitrarily chosen to assign three levels to each conversion process, but any decomposition is possible.

In MOSS, implementation of data-parallel reduction is relatively straightforward. We enclose the spectral data for each level of the atmosphere in an application-level object. Each of those objects is mirrored in some data parallel conversion process. The placement of the mirror objects determines where the computation is performed. Finally, the data conversion objects are themselves mirrored in the display process. This results in a data flow that is the same as the logical data flow shown in Figure 7.

However, in an event-stream-based monitoring implementation, such as Falcon or Progress, the results are quite different. When these systems provide for differential routing of events, it is typically only routing based on event type, not event contents. Because of this, the straightforward implementation of data parallel reduction leads to a data flow depicted in Figure 8. Here, each arrow again represents the transmission of data about a single atmospheric level, but

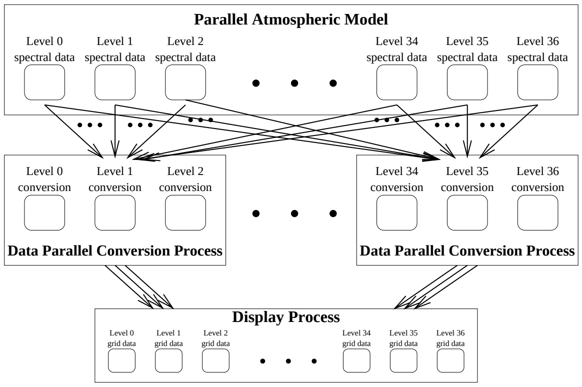

Figure 8: Event stream data flow in data parallel monitoring reduction.

the termination points of the arrows are more clustered to represent a single event-stream subscription for the process. There is no quantitative change in the final event flow to the display process because there is only one subscriber at that level. But the event flow from the atmospheric model to the conversion processes has increased dramatically. Because of the lack of data-driven routing, events associated with each level must be broadcast to all the conversion clients. These clients, who have knowledge of what levels they are assigned, then discard the levels that they are not interested in. However, the broadcast imposes a considerable cost in terms of network bandwidth, increasing the cost of the first level of data transfers by a factor of N, where N is the number of processes involved in the data parallel monitoring reduction.

On a Sun Sparc Model 170, the conversion of one level of atmospheric species data from spectral to grid form requires 2.5msec, consuming 253 complex numbers (about 2 Kbytes). If the conversion of all 37 levels were decomposed across 37 processors and run at full rate, MOSS would require a network bandwidth of roughly 30Mbps between the application and the conversion processors. An event-stream based system would require 37 times as much bandwidth, or over 1 $Gbps$. Conversely, if we were to assume that bandwidth were limited to 100 $Mbps$, MOSS could easily still support a 37 processor system which would process the model output at 400 Hz. However, an event stream-based solution would saturate the network at 11 processors, a level of parallelism which would result in a 100 Hz processing rate.

This analysis indicates that the event routine provided by the Mirror Object Model is an important contribution to the scalability of interactivity systems.

5.4 Automatic Data Suppression

In addition to appropriate data routing, data update suppression is an important technique for scaling an interactivity system. In order to demonstrate how important, we consider a different aspect of the data flow shown in Figure 7. In particular, consider the level of traffic between the spectral-to-grid conversion processes and the display process. Species concentration for an atmospheric level in grid form consists of 2048 floating point numbers, in this case about 8 Kbytes. At the fastest event rate possible with MOSS and the conversion parallelized across 37 processors, this amounts to 120Mbps. But in the distributed laboratory environment, there may be more than one display monitoring the application at a time. If each is sent the entire data set, they would *each* require 120Mbps of network bandwidth, enough to tax most networks. However, many displays will not use the entire 300Kbytes that each update will bring. Many displays are one- or two-dimensional and only need a particular column, row or level of atmospheric data. This indicates a need for a mechanism for specializing what data are sent based on the actual requirements of the data consumer.

This need has been recognized and met by different mechanisms in other systems. Falcon has a mechanism for enabling and disabling sensors. Paradyn inserts and removes instrumentation points as they are needed. However, most previous mechanisms are inadequate for the distributed laboratory environment. For example, the Falcon and Paradyn sensors essentially allow only that particular collection points be turned on and off. This type of mechanism is not easily adaptable to the situation in the previous paragraph, where we need to suppress update of parts of the atmosphere, but not all data of a particular type. Another difficulty with the simple on-off switches in systems like Falcon is that they do not work well when multiple data consumers are present. While each client knows what data it needs, no client has the global knowledge required to properly operate the switches. MOSS avoids this difficulty through the operation of event channels. Because there is one event channel per application-level object, and because the event channel properly handles multiple event consumers and suppresses network traffic when there are no consumers, the channels provide a natural update suppression mechanism that is data-specific and performs in a multi-consumer environment.

For example, in the situation described above, if each display is interested in data from only a single level and therefore creates a mirror object for only the level it is interested in, MOSS will transmit only the minimum 8Kbytes per update per display. Compared with almost 300K per update per display for a Falcon-style event-stream implementation. This represents a considerable savings in network load.

6 Conclusions

This paper presents the Mirror Object Model, a new higher-level model for programming distributed monitoring and steering systems for high-performance applications. Existing monitoring and steering systems suffer from a variety of problems. Many event-based systems have properties that prevent them from scaling to high event rates and large quantities of data. Additionally, no existing monitoring and steering systems provide a semantic model of computation that unifies application-level monitoring semantics, monitoring data reduction, and application-level steering enaction. The Mirror Object Model supplies the unifying semantic link between monitoring, externally triggered computation and steering enaction.

One of the principal results of this work is the Mirror Object Steering System (MOSS), a set of tools and libraries that provide an implementation of the Mirror Object Model. Sections 3 and 4 describe the Mirror Object Model and presents details of its realization in MOSS. MOSS is built upon several middleware packages including a distributed object system, a communications manager, and a binary I/O package. The performance analysis in Sections 5.1 and 5.2 show that MOSS provides the additional functionality of the Mirror Object Model without significant performance degradation. In fact, Section 5.1 shows that MOSS imposes considerably *less* overhead than other sampled monitoring mechanisms. Additionally, Section 5.3 and 5.4 show how the basic properties of the Mirror Object Model significantly reduce network overhead when scaling event rates through basic mechanisms such as data-parallel monitoring reduction.

The Mirror Object Model also allows the implementation of important optimizations which have the potential to open new ground to program monitoring and steering. Additionally, while this paper concentrated on scalability concerns, MOSS has significant advantages in programmability and in targeting the dynamic applications described in [15]. Future papers will explore these issues.

MOSS runs on a variety of platforms including Sun Sparc SunOS 4.1.3, Sun Sparc Solaris 2.x SGI MIPS IRIX 5.x, SGI MIPS (32 and 64-bit) IRIX 6.x, IBM RS6000 AIX 3.2, x86 Linux, x86 Solaris 2.x, and x86 Windows NT. Additional documentation and source for MOSS can be retrieved from: `http://www.cc.gatech.edu/systems/projects/MOSS/`.

References

[1] *CORBAservices: Common Object Services Specification*, chapter 4. Object Management Group, 1997. http://www.omg.org.

[2] Greg Eisenhauer. Portable Self-Describing Binary Data Streams. Technical Report GIT-CC-94-45, College of Computing, Georgia Institute of Technology, 1994. *(anon. ftp from ftp.cc.gatech.edu)*.

[3] Greg Eisenhauer. *An Object Infrastructure for High Performance Interactive Applications*. PhD thesis, Georgia Institute of Technology, June 1998.

[4] Greg Eisenhauer, Beth Plale, and Karsten Schwan. DataExchange: High Performance Communication in Distributed Laboratories. to appear in *Journal of Parallel Computing*, 1998.

[5] I. Foster, C. Kesselman, and S. Tuecke. The Nexus Approach to Integrating Multithreading and Communication. *Journal of Parallel and Distributed Computing*, pages 70–82, 1996.

[6] Weiming Gu. *On-line Monitoring and Interactive Steering of Large-Scale Parallel and Distributed Applications*. PhD thesis, Georgia Institute of Technology, Atlanta, GA 30332, 1995.

[7] Weiming Gu, Greg Eisenhauer, and Karsten Schwan. Falcon: On-line Monitoring and Steering of Parallel Programs. to appear in *Concurrency: Practice and Experience*.

[8] Rick Jones. NetPerf: A Network Performance Benchmark, Revision 2.1. http://www.cup.hp.com/netperf/NetperfPage.html, February 1996.

[9] 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.

[10] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The Paradyn Parallel Performance Measurement Tools. *IEEE Computer*, 1995.

[11] B. Mohr, A. Malony, and J. Cuny. TAU. In G. Wilson, editor, *Parallel Programming using C++*. MIT Press, 1996.

[12] Bodhi Mukherjee and Karsten Schwan. Experiments with a Configurable Lock for Multiprocessors. In *Proceedings of the International Conference on Parallel Processing, Michigan*, pages 205–208. IEEE, Aug. 1993.

[13] Beth Plale, Greg Eisenhauer, Jeremy Heiner, Vernard Martin, Karsten Schwan, and Jeffrey Vetter. From Interactive Applications to Distributed Laboratories. *IEEE Concurrency*, 1998.

[14] Daniel A. Reed, Ruth A. Aydt, Roger J. Noe, Phillip C. Roth, Keith A. Shields, Bradley Schwartz, and Luis F. Tavera. Scalable Performance Analysis: The Pablo Performance Analysis Environment. In *Proceedings of the Scalable Parallel Libraries Conference*. IEEE Computer Society Press, 1993.

[15] Daniel A. Reed, Christopher L. Elford, Tara M Madhyastha, Evgenia Smirni, and Stephen E. Lamm. The Next Frontier: Interactive and Closed Loop Performance Steering. In *Proceedings of the 1996 ICPP Workshop*, august 1996.

[16] Randy L. Ribler, Huseyin Simitci, and Daniel A. Reed. The Autopilot Performance-Directed Adaptive Control System. *Future Generation Computer Systems*, special issue (Performance Data Mining), submitted for publication, November 1997.

[17] Jeffrey Vetter. Computational Steering Annotated Bibliography. *ACM SIGPLAN Notices*, 36(6), 1997.

[18] J.S. Vetter and K. Schwan. Progress: a toolkit for interactive program steering. In *Proc. 1995 Int'l Conf. on Parallel Processing*, pages II/139–42, 1995.