Java Mirrors: Building Blocks for Interacting with High Performance Applications

Yuan Chen, Karsten Schwan and David Rosen

Abstract

Mirror objects are the key building blocks in the virtual 'workbenches' and 'portals' for scientific and engineering applications constructed by our group. This paper uses mirror objects in the implementation of the RTTB design workbench, which controls components of the RTTB rapid tooling and prototyping testbed. Mirror objects continuously mirror the states of remote software or even hardware entities, and the operations performed on mirrors are automatically propagated to these entities. Thus, end users perceive mirrors as virtualizations of remote entities. This paper presents the concept of mirror objects, their JMOSS Java-based implementation, the interoperation of JMOSS Java mirrors with the CORBA-based MOSS mirror object implementation, demonstrations of mirror functionality and utility with a virtual 'design workbench' used by engineers for rapid tooling and prototyping processes, and performance evaluations of mirror objects. We also present initial evaluations of JMOSS mirrors in mobile environments, where workbench users can continue their PC-based online interactions via handheld devices carried to the shopfloor.

1 Introduction

The Internet has created new opportunities for scientific collaboration. Scientists and engineers working in geographically different locations may use remote visualizations to access the results of their large-scale simulations, and they may update information gathered by remote instruments[20]. In addition, real-time collaboration tools enable them to evaluate and discuss their results and insights with remote colleagues [18], and they can even interact with their complex applications while they are running, perhaps simply to monitor their progress or to perform tasks like online program steering[24, 29].

Virtual 'Workbenches' for scientists and engineers. We are developing an interactive "Design Workbench" for the Rapid Tooling Testbed(RTTB)[26] being developed by researchers in Mechanical Engineering at Georgia Tech. The RTTB permits engineers to rapidly design, simulate, and prototype new mechanical parts using rapid prototyping manufacturing methods. The "Design Workbench" presents to such users an online virtualization of the RTTB, enabling them to (1) remotely interact with ongoing design simulations and with the software packages implementing those simulations, (2) utilize graphical interfaces to interact with instrumented portions of the physical RTTB (e.g., cameras viewing rapid tooling machines), and (3) use diverse interfaces ranging from web browsers or Java-based visualizations to high end 3D graphics displays that render in real-time the potentially large amounts of data being produced and evaluated. Moreover, such interactions may be maintained even as end users move from one interface or access device to another, perhaps initially inspecting an ongoing design simulation from their office machines, but then continuing their work while inspecting the associated prototype manufacture on the shopfloor, using handheld, wireless-connected devices. Finally, the design workbench assumes that simulations and prototyping processes are distributed and concurrent, the former typically comprised of multiple software components executing in parallel on multiprocessor or distributed machines, the latter involving multiple prototyping and manufacturing machines on the shopfloor. Other research groups have formulated similar characterizations for scientific or engineering 'workbenches' (sometimes also called 'portals') [17, 19, 5].

In comparison to remote inspection and instrumentation [20], an *interactivity system* like the design workbench puts equal weights on both the monitoring and control of its remote system. Specifically, the workbench must control the RTTB system's software packages, where computations may be started, paused, resumed, or stopped;

the parameter values being used may be changed and ongoing results captured. Such controls may be performed using computers located on the shopfloor or from a remote web browser. While an experiment is being performed, multiple engineers may jointly observe ongoing computations, not only with respect to their final results, but also including the outputs of intermediate computational steps. Based on the insights thus derived, engineers can steer their simulations by changing certain simulation parameters, then view the results of these changes and possibly, experiment with alternative parameter settings. The intents are to (1) remove the need for the physical presence of end users in the laboratory, (2) speed up the process of running simulations via which users experiment with different designs and design parameters, and (3) support the rapid construction of such experiments utilizing both commercial and experimental simulation components. More detail on the design workbench appears in Section 4, and an earlier implementation of the design workbench is described in [26, 12].

Mirror objects – building blocks for distributed workbenches and portals. This paper presents the design and implementation of *mirror objects*[6, 9], which are key building blocks used in the RTTB design workbench and in other interactivity systems for high performance applications constructed by our group. Mirror objects provide the basic functionality needed for workbench construction, in that they 'mirror' those behaviors of the target application being viewed or controlled via the workbench that are important to end users. Specifically, the state of a mirror object matches the state of an application component, because component updates trigger consequent updates of the mirror object's state. Furthermore, operations performed on the mirror object trigger updates to the application component that is being mirrored, thereby turning operations performed within the workbench into operations performed on the target application. Effectively, mirror objects are faithful virtualizations of applications components.

JMOSS Java mirrors. This paper uses the JMOSS Java implementation of mirror objects to implement the RTTB design workbench. We have also used mirror objects to monitor and tune high performance codes[9], and to implement scientific workbenches and portals, as described in [24] and in [30]. In ongoing work, mirror objects are being used to virtualize remote sensor devices in ubiquitous and embedded systems.

A previous publication by our group explains the design and implementation of the MOSS C/C++-based mirror objects, and their use for the online performance monitoring and steering of high performance, parallel and distributed codes[9]. This paper describes the JMOSS Java-based implementation of mirrors and their use for construction of realistic virtual workbenches. The main results presented in this paper (1) include performance evaluations that demonstrate the ability of JMOSS to mirror the state required for online control of the RTTB system via the design workbench, and (2) they show that the use of JMOSS mirrors exhibits overheads acceptable to high performance applications.

One advantage of Java (JMOSS) vs. non-Java (MOSS) mirror objects is their support of mobility. Such functionality makes it easy to construct mobile workbench components, where an end user can easily move his interactions from the desktop to a mobile device and back to the desktop, whenever needed. Three properties of JMOSS mirror objects are key to enabling such mobility. First, JMOSS mirrors may be moved at any time, by simple execution of a 'move' command, typically performed in response to some user action, such as initiating the 'handoff' from the desktop machine to a palmtop device. Second, JMOSS mirrors may be directly associated with certain application state, or they may mirror other MOSS or JMOSS mirrors, thus making it easy to define and refine the amounts of state (and its presentation) needed for handheld vs. other interface and access devices. Third, JMOSS migrations can be customized, where each migration may choose what internal state is migrated, thereby making it easy to move subsets of the state to a handheld device with a small screen vs. moving larger amounts of state to a handheld with a larger screen and more memory.Or, even when migrating to a handheld device of the same type as used previously, if its memory already contains other useful data or if migration must be performed quickly despite slow network connections, then the migration of a JMOSS mirror object to it may be customized to reduce bandwith or memory needs and migration delays. Measurements presented in this paper also demonstrate the utility of customized migration for the design workbench. For instance, we show that customized migration and mirroring enable real-time updates of mirror objects, compared to 'full' mirroring which results in delays for mirror updates exceeding 140ms for 100Kbyte data sets.

Hierarchical mirrors, mirror mobility, and customized migration are also useful for dynamic load balancing or to enable alternative representations and presentations of workbench components and data, making it easy for instance, to have simultaneous high resolution and wire-frame presentations of certain data, or to move a

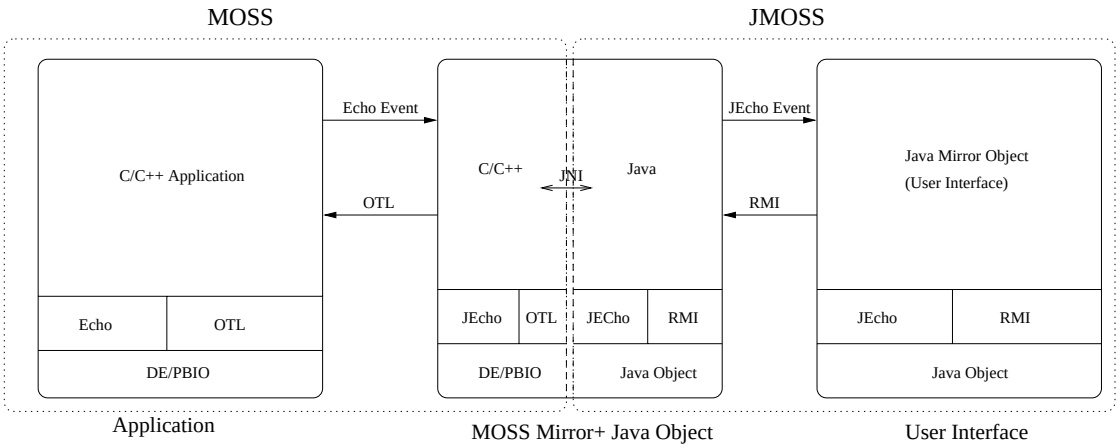


Figure 1: Mirror Object Architecture

rendering process from a highly to a lightly loaded machine. This paper does not explore these capabilities of mirrors in detail.

Interoperation of MOSS and JMOSS. In general, realistic engineering and scientific workbenches should offer both Java- and C/C++-based support. For the RTTB workbench, this implies the use of both MOSS and JMOSS in its implementation. Our motivation is twofold. First, the relative inability of Java to move large amounts of data prompts us to avoid using Java-based mirrors for certain workbench components, such as those that analyze the large outputs of the structural analyses performed for the mechanical parts being prototyped (e.g., the ProE finite element analysis code routinely outputs more than 100Kbytes of data even for small designs). Yet at the same time, second, it is important to permit workbenches to employ the wide variety of tools present in the Java domain, including the tools we use to generate 2D and 3D parts displays. These statements are equally true for the scientific workbenches we have constructed, which use Java-based tools like VisAD and Habanero[16] for smaller-scale visualizations and for collaboration, respectively, while also using non-Java tools like Vis5D or CAVE5D to create 3D or immensive displays of large data sets, or even using the parallel methods offered by POVRAY to perform high-quality rendering of such data in real-time. Given this variety of tools and implementation platforms, a useful workbench building block must support the mixed use of Java and non-Java tools. The MOSS/JMOSS mirror objects presented in this paper constitute such building blocks.

The overheads implied by interoperation of Java with non-Java programs are also evaluated in this paper. For example, we show that data conversion from raw data to Java objects is not overly expensive, typically adding less than 15% to data transfer costs. In addition, we show that the monitoring overheads implied by using JMOSS mirror objects are only slightly larger than those experienced by basic Java socket connections transferring the same amounts of data.

Overview. The remainder of this paper first provides an overview of the Mirror Object Model in Section 2. In Section 3, we discuss the key JMOSS concepts and their implementation. The use of mirror objects for constructing the RTTB workbench is demonstrated in Section 4. Performance evaluations and microbenchmarks appear in Section 5. Related work is discussed in Section 6, and Section 7 contains conclusions and future work.

2 The Mirror Object Model

Basic functionality. The Mirror Object Model views all application-level entities as objects with associated methods and state, even when the application is actually written in Fortran or C. To implement mirroring, we thus rely on the target applications' instrumentation. The results presented in this paper use manual instrumentation methods, but better methods are well-known, including source code annotations[9, 13], compiler and linker

support, or even runtime binary editing[14]. In any case, mirror objects are 'linked' to the target applications they mirror via asynchronous events updating their states in response to application-level state changes. Specifically, the JMOSS implementation of mirror objects generates for each state change in an instrumented application-level object a monitoring event, which is forwarded to the mirror object and updates its state. This process is asynchronous, in that application-level state changes are not delayed until mirror-level state changes have been completed. Our motivation is to avoid imposing unnecessary overheads on the high performance applications monitored and controlled by mirror objects.

The intent of mirror objects is to be faithful virtualizations of the application components they mirror. Thus, methods executed on mirrors that update their internal states must result in updates performed on application-level components. Such updates are implemented via synchronous remote method invocations that trigger 'steering' actions on application components. Details on program steering infrastructures and instrumentation appear in [29, 9]. For this paper, it suffices to state that an update operation on a mirror object is not complete until the steering action on the target application component has been performed. The purpose is to offer a simple update model to workbench users. In JMOSS, both the monitoring and the update consistency models may be adjusted for each mirror object, by changing its monitoring and update policies, respectively. Such policy changes are not explored here, but are discussed elsewhere in the context of useful consistency policies for distributed objects on shared and distributed memory machines (see [27]).

Mirror objects also contain the additional methods and/or derived state implied by the roles they play in virtual workbenches. They may offer certain state transformation of display methods, for instance, so that they render transformed and not raw, unintelligible application state[24, 30].

In general, then, the Mirror Object Model places interactivity and transformation methods and state into "mirror objects", requiring the application to be instrumented, but not requiring it to be enhanced with all of the additional state and methods implied by its interactive use. Mirror objects are analogues of the 'objects' contained in the target applications, sharing their state via monitoring and implementing their updates via remote method invocation.

Application programs may be mirrored many times and in many places, as per their instrumentation. In addition, a single application 'object' may be associated with any number of mirror objects, where by default, all mirrors observe the same application-level changes, and an operation executed on any mirror is synchronously reflected to the application-level object being mirrored. Thus, inherent in mirror objects is basic support for multiple observers and operators able to view the same data, each in ways customized to their individual needs. Not built into basic mirror objects are application-dependent update semantics, such as those that may require multiple mirrors to perform updates before the application-level object is updated. Finally, mirror objects themselves may be further mirrored, thereby creating hierarchies of mirrors and thereby, enabling the interactivity system to supply the application- and mirror-derived data they need to new objects and methods providing useful functionality.

MOSS and JMOSS mirror objects. Figure 1 depicts the architecture of MOSS and JMOSS mirror objects. MOSS is an implementation of mirror objects based on CORBA [9, 6] and supports both Fortran and C/C++ target applications.

JMOSS is the implementation of mirror objects realized in Java and evaluated in detail in this paper. JMOSS interoperates with MOSS mirror objects and/or with instrumented non-Java programs via a C-Java converter for the basic data types being exchanged between both. Monitored state of application objects required by mirrors is represented as typed monitoring events and propagated using the ECho and JECho event distribution systems, respectively, for MOSS and JMOSS objects. ECho and JECho, described in detail in [7] and in [30], implement the asynchronous transport of typed events via publish/subscribe communication infrastructures. Specifically, each mirror object subscribes to the event channels to which the application-level monitoring events they desire are sent. Thus, one mirror object can also receive events emanating from multiple application components, for instance to track timings or durations of certain application-level actions. Conversely, to propagate to application components the state changes resulting from operations on mirror objects, a MOSS mirror uses a thin object invocation layer termed the Object Transport Layer (OTL), whereas a JMOSS mirror object employs Java's remote method invocation (RMI) facility.

Figure 1 depicts a scenario in which a MOSS mirror object is further mirrored into the Java domain, using JMOSS. ECho events transport application-level updates to the MOSS mirror, and JECho events transport MOSS mirror updates to JMOSS mirrors. Performance evaluations appearing in Section 5 will evaluate the total delays and the data throughput experienced by this configuration. They will also evaluate the delays implied by JMOSS mirror object updates first propagated via RMI to MOSS objects and then propagated via OTL to application-resident objects. The versions of ECho and JECho employed in these experiments are those described in [7] and [30], using the underlying DataExchange and PBIO typed event representations[8]. Finally, it is the Java mirror object that either acts as or interacts with some Java- or browser-based user interface. We next describe those details of the JMOSS implementation required to understand its more advanced functionality, such as migration.

3 JMOSS Implementation

We have already explained how JMOSS/MOSS utilize the ECho and JECho publish/subscribe communication infrastructures for purposes of monitoring object state, and how updates on mirror objects are propagated to the application via RMI and OTL method invocations and via program steering functionality. Furthermore, for propagating updates on mirror objects to application-level objects, the lightweight OTL object transport layer are used by MOSS and JMOSS uses Java's standard implementation of RMI. Both ECho and JECho offer efficient implementations of event channels, but only JECho offers facilities for runtime migration, which are used in the JMOSS implementation of migratable mirror objects.

This section will describe the JMOSS implementation, specifically focusing on its additional functionality compared MOSS. We discuss interesting issues with the implementation and use of JMOSS concern the manner in which Java objects are instrumented, the interoperation of JMOSS with MOSS objects, Java mirror migration, the customization of such migration and customized mirroring. We also discuss implementation choices and possible improvements.

The remainder of this section explains sample mirror objects and their implementation in more detail, using the RTTB application as an example.

Interoperation of MOSS and JMOSS Objects. MOSS and JMOSS objects interact via the Java native interface. Specifically, a MOSS mirror produces events described in some standard native form. These events are received by the MOSS native library linked with the JVM, then converted to Java objects, followed by the invocation of appropriate JECho-provided Java event delivery methods. JMOSS's Java/native translation library translates Java objects to/from native data represented with the PBIO binary data format. Conversely, a Java mirror object interacts with a MOSS object by calling native procedures in the MOSS library. **Java Object and JMOSS Mirrors** . Now we describe how to make an existing Java object available for steering and monitoring and how to create its mirror object. The interactions of any JMOSS mirrors with a Java object are controlled by a 'wrapper' object. Each such wrapper controls access to the internal state of the Java object. Such state is read by calling the wrapper's 'get attribute' methods, and it is updated with 'set attribute' methods. The wrapper also propagates state updates to Java mirror objects, by creation of appropriate event channels, subscription to such channels as a provider, and publication of the state to the JMOSS mirrors that have subscribed to the channel. Finally, the wrapper handles invocations on Java application object by updating its internal state and invoking the proper methods on Java application object. Figure 2 shows the relationship between Java application object(Java object corresponding to MOSS mirror in our object model),Java wrapper object and JMOSS mirror object in interactive program.

JMOSS provide a tool called *jmossw* wrapper generator assists users in the creation of wrappers for existing Java object. With the help of *jmossw*, to create a wrapper, users:

1. Define JMOSS mirror interface in Java, which is a remote interface that extends java.rmi.Remote;
2. Define `State` class, which defines the interesting state to be contained in the Java application object;
3. Use *jmossw* to generate a wrapper class from mirror interface and `State` class; and

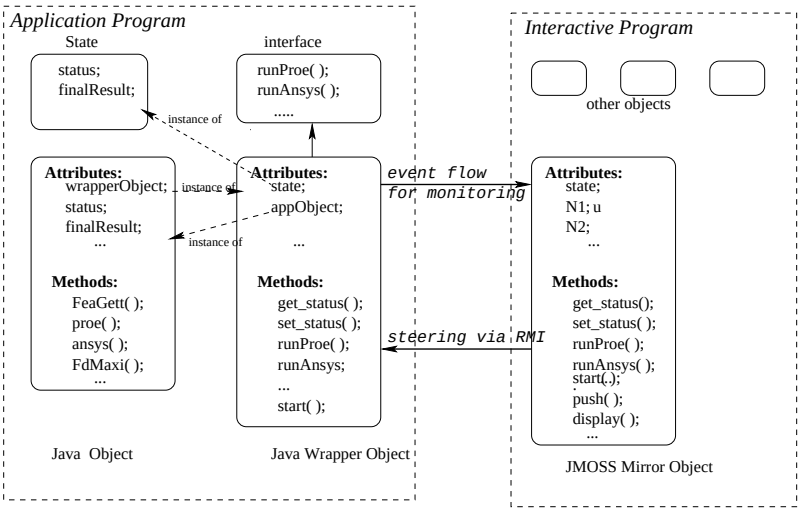


Figure 2: Wrapping Java object for monitoring and steering

4. Supply implementations of the 'get' and 'set' attribute methods to be used by the wrapper class, as well as perform any additional modifications of the wrapper class that may be required.

A sample code segment for the RTTB design workbench, including the mirror interface and the `State` class defined by the user, and the wrapper class generated by *jmossw* is shown in shown in appendix A.

JMOSS mirror object receives application's state updates by subscribing to appropriate JECho's channel created by the wrapper. The steering operations are accomplished by invoking the wrapper object's methods using RMI. JMOSS provides another tool *jmossm* to help users write interactive programs from scratch. *jmossm* is a mirror object generator which creates JMOSS mirror object for users. A sample JMOSS mirror object `MirrorExperiment` generated by *jmossm* can be found in appendix B.

With the help of the *jmossw* and *jmossm* generators, the construction of portals or workbenches is rather simple. The original application in c/c++ is properly instrumented. MOSS mirrors for the application are created using MOSS. Java object corresponding to MOSS mirror is created. The user defines the Java interface and uses *jmossw* and *jmossm* to generate Java wrapper and Java mirror. Program monitoring and steering are performed by inspecting Java mirror objects and calling their methods.

Discussion. Several comments may be made about JMOSS mirror objects and their functionality. First, extending the functionality of MOSS mirror objects, JMOSS mirrors permit developers to vary the ways in which state monitoring and state updates are propagated, using either synchronous or asynchronous methods. Asynchronous methods are particularly important when large numbers of updates are made or when perturbation on the target application must be small. Measurements contrasting synchronous with asynchronous event delivery appear in [30].

Second, method calls on mirror objects are defined for each attribute contained in its mirror state. Specifically, for each attribute, `get` and `set` methods are available. Their execution results in the inspection and return of mirror object state or in the execution of an RMI call that updates the state of the object being mirrored. Additional methods for mirror objects may be explicitly defined by workbench developers.

Third, to implement efficient data transfer across the MOSS and JMOSS non-Java vs. Java execution environments, we use a Java-C converter and Java's JNI interface. to support the efficient conversion between the binary data representations being used (i.e., encoded with PBIO[8]) and the Java objects corresponding to them. This enables JMOSS to better address the real-time operational needs of the high performance applications it virtualizes. Performance measurements in Section 5 demonstrate the efficiency of Java to non-Java communications.

Finally, by using wrapper classes and providing tools like *javaw* and *javaw*, we separate the Java wrapper from Java application and minimize the modification of existing Java objects.

3.1 Advanced JMOSS Functionality

Customized Migration. JMOSS mirror objects can migrate while in use. Such migration is implemented using the *MOBject* lightweight object migration system, which is also used by JECho when event consumers or producers migrate while event channels are in use. *MOBject* is based on the Mole[28] system for Java agents, but offers reduced migration overheads compared to Mole. *MOBject* is lightweight in that it does not use object serialization or RMI to migrate objects, whereas Mole uses both for agent migration. Instead, *MObject* migrates objects with JECho's optimized object streams and `send()` methods. The JECho object stream is a simplified version of Java's standard object stream, offering cost savings of up to 71% compared to the regular stream for object serialization or deserialization time.

While JMOSS mirrors migrate, additional updates to be received by them are not lost. Instead, the event channel that *feeds* updates to the mirror buffers and then produces all events sent to it, in the same order observed if migration had not occurred. The details of this implementation are described in [30].

Important to mobile workbench components is JMOSS' ability to customize migration, by variation of the amounts of state transported from the 'old' to the 'new' object. The *MOBject* object migration facility implements this functionality as follows. First, when an object is ready to migrate, it calls the `migrateTo(location)` method. Second, after `location` has approved of the migration, the original object copy is suspended by suspending all of its threads. Third, the original mirror object is serialized, including the objects it references. Fourth, the serialized object is sent to the destination location using a JECho `send()` operation. Fourth, the object is reconstructed at the destination and resumes its execution. In this process, *customized migration* involves (1) customizing the amounts of state transferred during migration, (2) controlling the ways in which state is restored at the target, and (3) changing the target object's behavior in comparison to the original one. Concerning (1), users may declare only certain object fields to be serializable. Fields not declared as such are ignored during migration. The declaration:

```
private ObjectStreamField[] serialFields;
```

contains the names and types of all fields to be serialized during migration (by default, 'serialFields' contains all fields in the class being migrated). Concerning (2) customization is possible during deserialization, where during object initialization, for instance, user-provided procedures may recompute the values of fields that were not migrated. Finally, (3) the *MOBject* migration facility permits the class of the target object to differ from that of the original one, so that the new object may use implementations of methods better matched to the migrated object's new tasks (e.g., for rendering data on a small handheld's display).

Before MObject starts real migration, it will call the method stop( ) and give the object a chance to dispose of any objects he doesn't really need. Thus, the user may define serialFields in the method stop( ) in order to customize the serialization. serialFields itself and objectName must be serialized. When the serialized stream reaches the destination, the object location first deserializes to get the values of serialFields and objectName. According to the object name, the location creates a new object or retrieves from somewhere, such as a code repository. According to the fields stored in serialFields, MObject can deserialize the left stream and assign the value to the fields with the same names as the source. Since some fields at the destination may be recomputed from other fields, the user can customize the deserialization by defining a reconstruction procedures in the method prepare( ), which is a function called before the agent start running. For example, if there are three fields low, high and average in a class. average = (hight+low)/2. We only need to serialize low and high during migration. After it reaches the destination, we can recompute the average from low and high. The code to do the recomputation should be put in the method prepare( ). Thus, JMOSS realized customizable migration by customizing serialization and deserialization. The different version of some object is specified by the variable objectName defined in the class *MObject*, which is moved with the Object. The mobile object system will

create an instance specified in the variable when it reaches the destination. The class can be loaded locally or downloaded from a remote repository by a class server.

Customized Mirroring. Customized migration permits us to change the amounts of state migrated and the ways in which state is initialized and used at the target. In contrast, *customized mirroring* controls how state updates on mirror objects are performed. This is important because mirrors are updated continuously. , in accordance with changes in the target application being mirrored. Thus, Customizing this update process can have significant effects. For instance, rather than sending two values to the mirror only to then compute their average in the mirror and discard the originals, customized mirroring permits us to 'move' at runtime this averaging computation from the mirror to the object being mirrored, thereby reducing communication overheads at only slightly increased perturbation in the target application.

Customization of state mirroring is critical to JMOSS's ability to deliver suitable performance for the heterogeneous underlying hardware used by the RTTB design workbench. JMOSS' customization of mirroring uses low-level support for handler migration offered by the JECho event transport facility employed by JMOSS. To use such support, JMOSS allows each mirror object to define a 'policy object' that controls how state information is transmitted from the target to the mirror object. This policy object is the entity being moved by JECho's lower level support for event channel customization. This support, termed 'eager handlers', in described in detail in [30].

JMOSS accomplishes customizable mirroring by employing JECho's eager handler. Eager handler permits an event consumer to specialize the content and the manner of handling and delivery of events by producers. This is achieved by 'splitting' the consumer's event handler into two parts, with one part remaining in the consumer's space and the other part replicated and sent into each event supplier's space. We term the latter event modulator while the other part that stays local to the consumer event demodulator,as events now go through the remote part of the handler first then travels through the wire and handed to the local part. JMOSS's policy object is implemented as a modulator. The policy object is specified when creating the consumer handle. A policy object is modulator which specifies its response to relevant state changes occurring at the supplier by defining intercept functions. One of the intercept function is Enqueue method. Enqueue is invoked at the time a producer pushes an event onto the channel. The method can perform any operation on the event, including discarding and transforming the event. Through its policy object, a mirror object can control if, how or when the updates are sent. The policy object can be dynamically changed at runtime. pch.reset is used to set the new policy object. The following code segment demonstrates a policy object which specifies that the mirror object is only interested in the final result. Customizable mirroring enable the mirror object to dynamically control propagation of application state in accordance with its interest and resource availability.

```
public class PolicyObject extends FIFOModulator {
  public void enqueue(DECEvent e) {
       State state = (State)e;
       if(state.status == DONE)
         super.enqueue(e);
       else
         return;
  }
}
```

4 The RTTB Design Workbench

This section demonstrates the usage of mirror objects for construction of a design workbench used by Mechanical Engineers. The workbench and the physical machines associated with it are termed the RTTB Rapid Tooling Testbed.

4.1 The RTTB Testbed

The RTTB testbed is intended to be a distributed computing environment to support product design, prototyping, and manufacturing[25]. The engineers using it are experimenting with different design processes and different sets of tasks, personnel, vendors, software, and equipment. As part of the RTTB effort, multiple computing infrastructures have provided communication, information sharing, work-flow, and distributed computation capabilities[11]. The design workbench described in this paper constitutes one such infrastructure.

Rapid prototyping and tooling. The RTTB addresses a set of emerging and commercial rapid prototyping and rapid tooling technologies for producing parts[26]. It utilizes both commercial and research software applications for design, including CAD systems like ProEngineer (ProE), SolidWorks, and CODA, synthesis tools like DSIDES and OptdesX, analysis tools like the ANSYS finite element analysis package used in this paper[2], manufacturing process simulation methods like MoldFlow and CMOLD, and process planning (e.g., CABSS) and selection tools. A typical usage scenario involves a designer designing a product, such as a cell-phone, and needing a set of prototypes of the product's housing for functional tests. Depending upon the number of prototypes needed and the testing requirements, RP technologies (such as stereolithography or selective laser sintering) may be selected, or a rapid tooling process may be selected in order to fabricate a larger number of housings. If a small number of prototypes is needed, a RP technology will be selected, along with a suitable material. If rapid tooling is used, then a particular process must be selected, a mold designed and fabricated, and the parts injection molded. Rapid tooling refers to a class of manufacturing processes that involve fabricating molds for injection molding. These molds may be built on a RP machine, they may be cast using patterns produced on a RP machine, or they may be machined. Since there are alternative rapid tooling processes, several different people or organizations, may be involved. For example, a rapid tooling process may involve the part designer, a service bureau that manages the manufacturing process, a mold designer, a mold manufacturer, and an injection molder. The decisions necessary to manufacture and deliver quality parts to the designer will be made by some combination of these people and organizations.

Using the RTTB Workbench. To understand how a distributed computing environment like the RTTB design workbench can aid RTTB users, consider the potential interaction among the parts,mold designers and the service bureau manager. Assume that a designer has an initial housing design and needs 20 prototypes for durability and drop tests. After identifying three possible rapid tooling processes, by describing his situation to the service bureau manager, they agree on three candidate rapid tooling processes. the designer uses the RTTB workbench to start a multiobjective optimization code, to fine-tune his design and incorporate manufacturing and material characteristics into the design process. In each optimization loop, finite element analyses are required for strength determination and injection molding simulations are required (each of which requires several hours of computation time). As the optimization progresses, the designer and manager are monitors its progress and perhaps, also steers it using RTTB mirror objects. At a point in time, the manager designer realizes that an alternative rapid tooling process becomes favorable, but needs to check with a third party. another vendor in another city. He thus directs the RTTB workbench to mirror his view of the computation status to the third party, but the vendor to get his input. The RTTB tailors this view as appropriate, to suit the third party's computational capabilities and to hide proprietary information, for example. that is not relevant to the vendor's expertise. The vendor then checks the suitability of his equipment for this application and provides manufacturing process capability information to the product designer, so that the designer's optimization process can be fine-tuned to suit this particular vendor.

Actions like those described above use the full complement of RTTB workbench functionality, including dynamic mirror object creation and deletion, their migration and customization, and continuous application monitoring and control via mirrors. Mirror objects have to interact with both research and commercial codes. In summary, the needs regarding distributed computations for the RTTB include wrappers for commercial and research codes, coordination among these codes and humans, information/document sharing, custom mirror, and custom migration.

Sample RTTB configuration and use. The more specific RTTB configuration shown in Figures 3 and 4 is the basis for some of the experimental evaluations performed in Section 5. As illustrated in these figures, this configuration of the RTTB is divided into three phases, normally associated with the product design timeline,

Figure 3: The Rapid Tooling TestBed

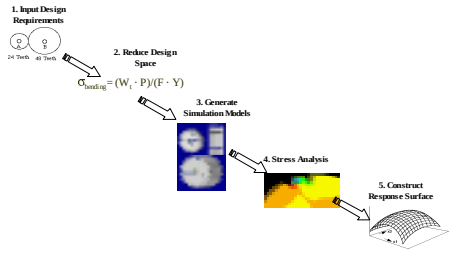


Figure 4: Gear Design Process in RTTB

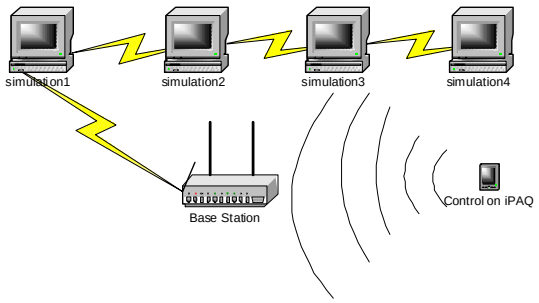


Figure 5: RTTB Application in Wireless Environment

which is comprised of which are design, design for manufacture, and manufacturing. During the design phase, requirements are entered into the testbed. These requirements consist of a part representation along with the design specifications for the part. In the DFM stage, suitable part and mold materials, rapid prototype technologies, and injection molding technologies are selected for the fabrication of both the injection molding tooling and the part. In addition, the injection molding tooling is designed and the part is tailored to facilitate the manufacturing of both the mold and part. The final stage involves the actual manufacturing of both the tooling and the part.

The scenario shown in these figures concerns the design of a spur gear, given a set of design requirements, such as gear ratio, transmission power and turning speed, etc. The designer has to present an optimal preliminary design by constructing a response surface of the maximum stress on the gear teeth vs. different gear dimensions. Figure 4 depicts this design process. In step 1, the design requirements are evaluated, and the designer presents a basic design blueprint. Then in step 2, the design space is reduced by calculating the maximum bending stress using engineering formulation. In order to construct an accurate response surface of the maximum bending stress vs. dimensions of the gear, an experiment is designed. 33 gears are selected within the design space and each of them is analyzed using Finite Element Analysis software for the maximum bending stress on teeth. Therefore, in step 3, an experiment is designed, including 33 samples. The CAD models of the samples (gears) are constructed using the ProEngineer (ProE) software package. Each gear is then analyzed using the AnSys package, in step 4. Finally, the maximum bending stress of each experiment sample is calculated, the response surface is constructed, and the optimal gear dimensions are decided by selecting one point on the response surface with the minimum bending stress. **Experiment Details.** The simulation experiment including steps 3, 4 and 5 is what we explore in this paper. In order to construct an accurate response surface, 33 samples are selected within the design space. And each sample (a gear design) takes 5 to 7 minutes to be constructed in ProEngineer and analyzed in Ansys. Therefore, a significant time savings for the designer can be achieved by running the experiment in parallel and allowing the designer to inspect the on-going simulation results. That is, after the experiment is designed, the samples can be parsed, and each sample can be executed in one computer.

In this experiment, four software packages are run to analyze a sample, the experiment designing and parsing package FeaGett, Geometric Modeling package ProEngineer, the Finite Element Analysis package Ansys, and the response surface calculating package FdMaxi. FeaGett and FdMaxi are locally developed research packages. System Realization Lab,Georgia Institute of Technology. A typical sample procedure involves generating a trail

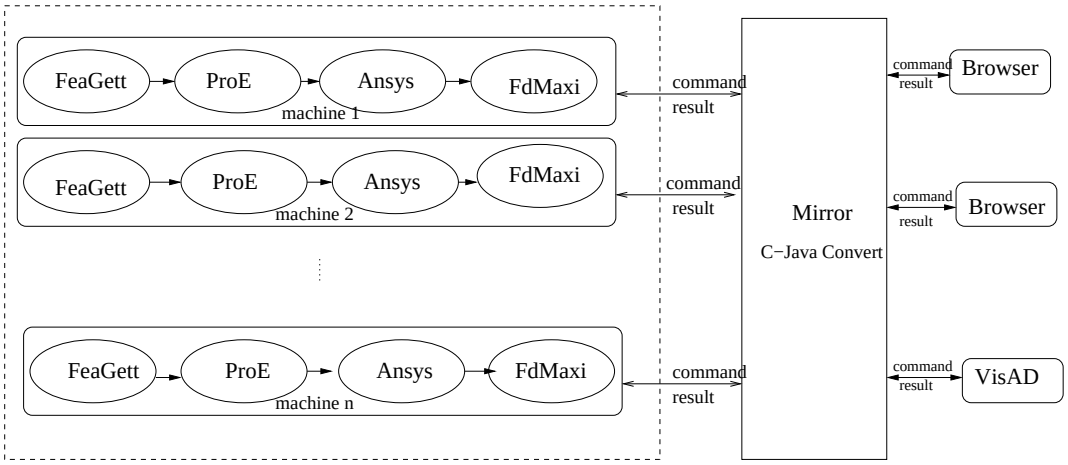


Figure 6: RTTB workbench

file from a parameter file (FeaGett), executing the simulation program (ProE), then the analysis program (Ansys) and computing the final result (FdMaxi). In our current implementation, all components used in a sample are 'wrapped' into an object containing all relevant attributes and status information. The JMOSS system operates the wrappers that further manipulate the commercial software packages.

This experiment executes multiple simulations and analyses in parallel..Multiple simulations run on workstations connected via Ethernet. Researchers use Linux iPAQ pocket PC to control and monitor ongoing experiments via wireless network. Figure 5 shows the environment. For each sample we have a set of attributes, including input parameters, intermediate results, final result, current status and a unique id. The user creates a mirror object, through which end users can initiate certain activities, monitor and control their execution, and even change parameters in the ongoing experiment. The user observes the ongoing computation at each of the nodes, not only the final result, but also intermediate steps. Full control over the computation is also enabled: the computation can be started, paused, resumed, or stopped, parameter values can be changed, for which both local and browser-based interfaces may be used, both of which employ mirror objects. Since intermediate results are typically quite large, engineers are interested only in viewing small subsets of these results, again using mirror objects and thereby, reducing the overheads implied by dynamically viewing such data.

Outcomes and Discussion. Figure 6 depicts the RTTB workbench architecture built using MOSS and JMOSS, including the parallel nature of ongoing experiments, the mirroring targeting multiple end users and user interfaces, and the online control exerted via those interfaces. This picture and this section's explanations indicate that the use of mirror objects for RTTB workbench construction has several desirable outcomes. First, it removes the need for the physical presence of end users on the laboratory computers where certain software packages are installed and available, while still enabling them to both view and control ongoing computational processes. Second, the multi-source, multi-target nature of mirror objects makes it easy for one designer to view multiple aspects of one experiment and for multiple designers to collaborate on a single ongoing experiment. Third, a single designer can even start and control multiple instances of an experiment, thus using parallelism to speed up the design process. Fourth, experiments may be constructed using both commercial and research software, and can even include remote views of camera images centered on certain production machines. Finally, while not done in the sample scenario shown in Figure 6, the mirror objects used within the workbench may be migrated to remote sites or machines, even while experiments are ongoing, thus giving designers considerable freedom to interact with collaborators or remote vendors and with the physical machines involved in rapid tooling or prototyping processes.

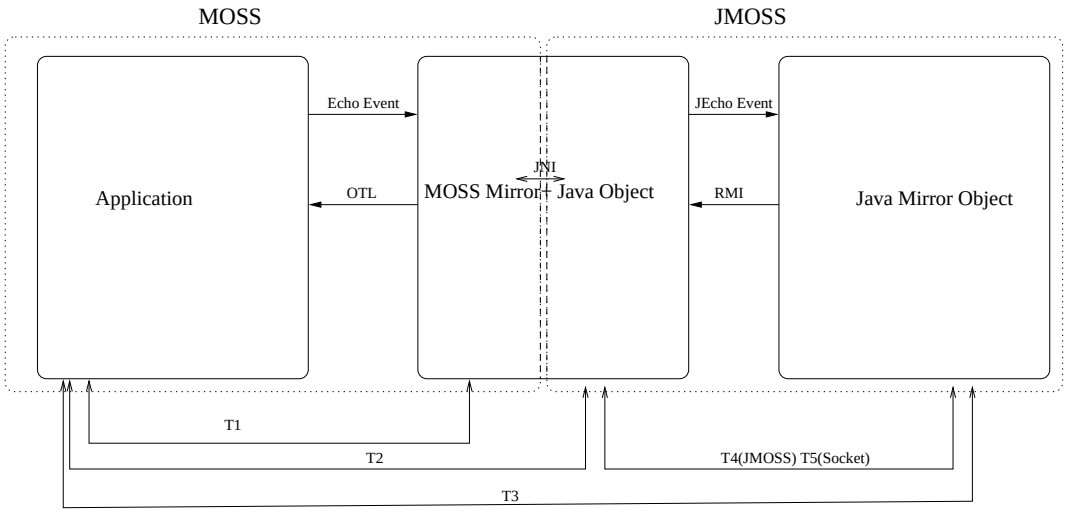


Figure 7: Experiment Configuration

5 Performance

5.1 Basic Benchmarks

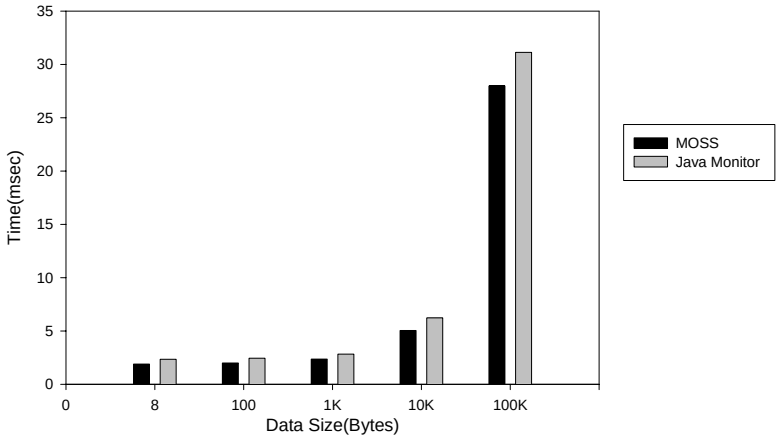
The purpose of the results shown in this section is to establish the basic performance of the MOSS/JMOSS systems, in part to demonstrate that mirror objects constructed with them are viable means for mirroring and controlling high performance applications like those occurring in the RTTB testbed.

Figure 7 depicts the basic software configuration used in all experiments, involving a target application component, a MOSS mirror, and a JMOSS mirror. The times measured are labeled with T1-T4, and the figure also shows the software being exercised, including the event transport systems ECho and JECho and the remote object invocation methods OTL and RMI. All measurements are performed on three UltraSparc Stations (Ultra 30) running Solaris 2.7, connected by 100Mbps Ethernet.

Basic measurements. We first evaluate the time required to complete a roundtrip through a MOSS mirror and ending up in a Java object using the JNI interface (see Figure 8. This represents the minimum delay one could experience when viewing and controlling a RTTB testbed software component via MOSS from some Java-based interface. Each test is comprised of a user-level Java program initiating some steering action, communicating his request to the application via JNI and OTL, then executing the requested state changes in the application and finally, sending the updated state information back to the Java program via ECho and the C-Java converter.

When executing a steering procedure in the target application, and when varying both the sizes of steering parameters from Java client to application and the amounts of state mirrored from application to Java client, results depicted in Figure 7 demonstrate the following. First, basic round-trip costs are roughly 2 milliseconds, and they increase significantly only when data sizes (the data used is an array of floats) exceed 1Kbytes. Second, C-Java conversion costs are acceptable for the relatively simple array data structures used in these tests, adding no more than 20% to the total costs of such a round-trip (e.g., consider the table entry for data of size 10K, where conversion costs add 1 millisecond to the 5 millisecond delay experienced by the MOSS mirror). For complex nested data structures being transported, conversion costs tend to increase by another 10-20%.

Comparison to Java sockets. A second set of tests measures the end-to-end delay for a JMOSS mirror object, using RMI to propagate an operation on the JMOSS mirror to the MOSS mirror, then using OTL to propagate it to the target application component. Monitoring entails the use of both ECho and JECho, including C-Java conversion. We again vary the amounts of state being mirrored and transferred from the application to the JMOSS object in response to each steering action, and we also compare to these measurements the costs of using a Java socket to communicate between the MOSS and JMOSS mirror objects. Our intent is to demonstrate that



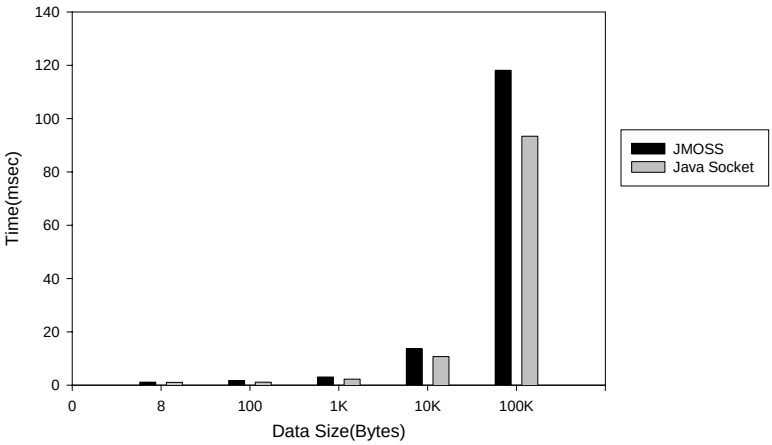
Data Size(Bytes)	MOSS(ms)	MOSS+Conversion(ms)
10	1.8857	2.3304
100	1.9945	2.4499
1K	2.3577	2.8180
10K	5.0484	6.2166
100K	27.970	31.118

Figure 8: Elapsed real-time for MOSS(T1)and MOSS+Conversion(T2)

JMOSS performance does not much exceed that of basic Java communications. The results shown in Figure 9 demonstrate that JMOSS round trip time is comparable to Java socket communication delays, including for large data transfers.

The difference between JMOSS and sockets is due to two factors. First, both RMI and JECho are built on top of Java Sockets, thus adding delays. In addition, a JMOSS mirror object has to wait for the return of a remote call before initiating a second call, since RMI call is a synchronous operation. When using our own asynchronous version of RMI, JECho's performance is quite close to the performance of Java sockets, as shown in [30]. This demonstrates that more efficient RMI implementations[15] will substantially improve JMOSS performance. It also shows that changes to update semantics for MOSS or JMOSS mirrors can substantially improve mirror object performance. The utility of such changes depends on the ways in which mirrors are used, however.

Migration Costs. Figure 10 shows the basic costs of mirror object migration, which include the cost of migrating the mirror object itself and of migrating the event channel associated with it. In the first setup, the object is migrated across two workstations connected with via 100MB Ethernet. The second setup uses one workstation and one iPAQ H3650 handheld device. The iPAQ is connected to the workstation via a IEEE 802.11b wireless network. Again, we vary the amounts of object state being migrated. Results show that migration costs strongly depend on the amounts of state migrated. They also show that migration cost can be quite high, especially when using wireless networks. This motivates our work on customized mirroring and migration, using which the amounts of state mirrored or migrated can be reduced. **Customized Mirroring.** Our final microbenchmarks evaluate the utility of custom mirroring, using end-to-end measurements between a JMOSS Java mirror object and a native program. The times shown are delays of MOSS, MOSS+conversion and JMOSS(the sums of MOSS, data conversion, and J delays) for different sizes of data. These results indicate that JMOSS mirror objects are somewhat impractical for mirroring large amounts of state(JMOSS's latency is more than 5 times larger than MOSS for 100KB data), thus arguing for customizing mirroring to suppress undesired state information. A simple demonstration of customized mirroring used in the performance results depicted in Figure 12 is one in which only 10% of total state is mirrored from the MOSS mirror to the JMOSS object. This results show that JMOSS's response times after customized mirroring are comparable to MOSS. This results in a reduction of total mirroring delay from approximate 150 to 30 milliseconds. For design workbench end users, this would mean the difference between receiving what appears to be non-real-time vs. real-time service. **Mobile Mirrors.** Mobile mirrors operate in an environment where network latency is high, bandwidth is low, and connections may



Data Size (bytes)	JMOSS(ms)	Java Sockets(ms)
10	1.1075	1.0159
100	1.6967	1.1563
1K	3.0239	2.2771
10K	13.6005	10.6801
100K	118.1345	93.3992

Figure 9: Elapsed real-time for JMOSS(T4) and Java Socket(T5)

Data Size (bytes)	Wired(ms)	Wireless(ms)
1K	73.202	100.013
10K	105.139	786.945
100K	262.811	4204.761

Figure 10: Migration Time

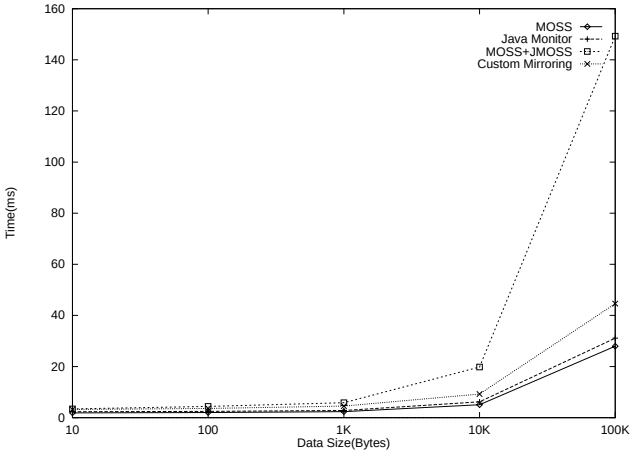
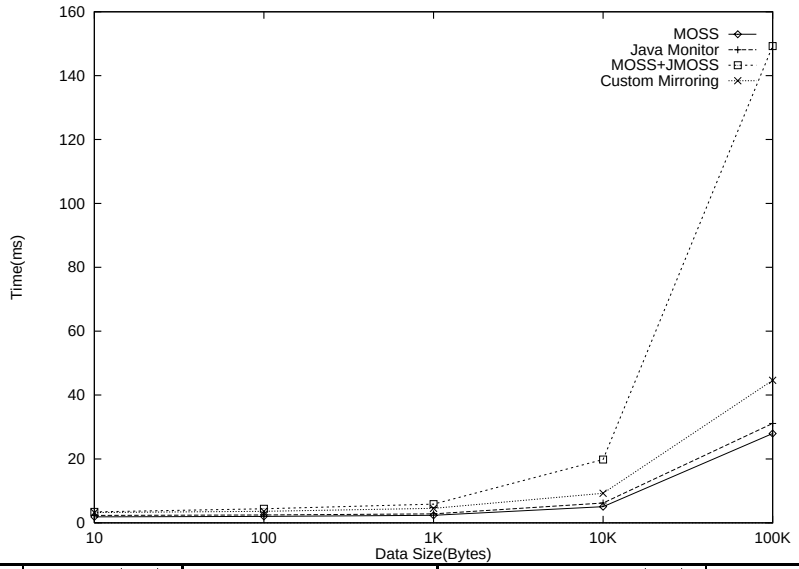


Figure 11: Elapsed real-time of MOSS(T1), MOSS+Conversion(T2), JMOSS(T3)without Custom Mirroring and JMOSS(T3) with Custom Mirroring



Data Size(bytes)	MOSS(ms)	MOSS+Conversion	MOSS+JMOSS(ms)	Custom Mirroring(ms)
10	1.8857	2.3304	3.4379	3.2242
100	1.9945	2.4499	4.1466	3.6026
1K	2.3577	2.8180	5.8419	4.5524
10K	5.0484	6.2166	19.817	9.25131
100K	27.970	31.1180	149.2525	44.6286

Figure 12: Elapsed real-time of MOSS(T1), MOSS+Conversion(T2), MOSS+JMOSS(T3) and MOSS+JMOSS(T3) with custom mirroring

Data Size (bytes)	Wired(ms)	Wireless(ms)
10	1.108	5.389
100	1.697	12.971
1K	3.024	84.299
10K	13.601	789.294
100K	118.135	7743.617

Figure 13: Elapsed real-time of JMOSS on Wired vs. Wireless networks

be intermittent. A simple wireless-connected mirror object is depicted in Figure 13. The figure also lists the basic costs experienced by JMOSS objects when using a wireless network. In these measurement, the application object is running on a workstation, and one mirror object resides on an iPAQ H3650 running the Linux operating system. 802.11b wireless LAN communication devices are used.

In comparison to the latencies experienced over a wired network, shown in Figure 9, our results show that the latency over the wireless network can be more than 70 times larger. In our configuration, this increased latency is due to the limited bandwidth available in the wireless domain (802.11b WaveLAN devices offer a maximum of 11MB/sec bandwidth, but the effective bandwidth achieved in our lab, due to interference and shared use by other devices, is typically no more than 300Kbytes/sec). In environments like these, functionalities like customized mirroring and migration are critical for achieving what appears to be real-time, interactive mirroring to end user.

Benefits attained from parallel tool execution. Benefits derived from the embarrassingly parallel execution of multiple instances of the same simulation are well-understood. We will not reproduce the performance results attained in this fashion. It is interesting to note, however, that the parallel execution of multiple simulation instances need not lead to a corresponding linear decrease in the total design time experienced by the parts

designer. This is because new settings for design parameters may be based on the results of previous experiments. This issue is being studied further by the Mechanical Engineers running the RTTB project.

5.2 Summary and Discussion

The MOSS and JMOSS implementations use well-optimized implementations for the event-based transport of state mirrored from application to mirror objects (i.e., ECho and JECho have been well-tuned, as described in [30] and in [7]). This is because mirror updates tend to be frequent, because mirrors should appear to end users as if they where in fact, certain target application component(s). However, neither OTL nor RMI have been optimized, so that experiments in which large state must be transferred from the mirror to the application exhibit low performance. While we have not yet experienced design workbench usage requiring higher RMI or OTL performance, ongoing work in our group is addressing these issues.

Benchmarks also demonstrate the importance of customizing mirroring and migration, in both cases reducing data transfer needs and therefore, significantly reducing the latencies of such actions.

6 Related Work

The high performance computing community has undertaken multiple efforts to understand and evaluate the utility of object technologies for high performance, distributed applications. The Diesel Combustion Collaboratory (DCC)[21] was a pilot project to develop and deploy collaborative technologies to combustion researchers distributed throughout the DOE national laboratories, academia and industry. The result was a problem-solving environment for combustion research. Similarly, the DCC selected the Product Realization Environment (PRE), a CORBA-based framework that shields software developers from raw CORBA, and its vendor implementation as a software architecture for tool integration and data exchange. Compared with the DCC, JMOSS provides additional support for mobility and for customizing mirroring and migration. In addition, mirror objects appear to match well the functional requirements of workbenches that virtualize remote software and/or hardware components. Security is a key aspect of the DCC; we do not address this issue.

Deepview[23] is a service-based framework for microscopy that is distributed, extensible, and maximizes the uses of common off-the-shelf software. It uses a standard CORBA object system implementation. In contrast to CORBA-based implementations of functionality akin to what is offered by mirror objects, MOSS demonstrates substantially better performance for state mirroring[7, 9]. JECho and thus, JMOSS also demonstrate improved performance for state mirroring compared to other Java-implemented event systems. Furthermore, we support the efficient integration and use of both Java and non-Java components into workbenches. Other systems offer less efficient methods for such interoperability via IIOP.

Other environments built on top of CORBA include JACO3 (Java- and CORBA- based Collaborative Environments for Coupled Simulation) and Webflow from Syracuse[1]. Compared with JMOSS, higher levels of skill and working knowledge of CORBA are required by programmers who wish to integrate a tool or application into these systems.

The Common Portal Architecture (CPA)[3] is a common component architecture specification, with a reference implementation described in [4]. The philosophy of CCA is to precisely define the rules for constructing components, to specify the required behavior a component must exhibit, and to define the interfaces between components and the framework. The goal is to provide standard ways of building components that are easily reused in CCA-compliant frameworks, such as the portal architectures being designed for high performance applications[17, 19]. MOSS and JMOSS are not CCA-compliant, but the functionality they offer must be present in most of the workbenches and portals currently being developed. Thus, our work provides insights into what constitutes useful CCA functionality.

Cactus[10] is an open source problem solving environment designed for scientists and engineers which also supports remote monitoring and steering. It enables parallel computation across different architectures and

collaborative code development across multiple users. Cactus uses thorn http to permit it to act as a web server, to change steerable parameters and to query information about an ongoing experiment. Compared to Cactus, MOSS and JMOSS offer higher performance and additional support for efficient object monitoring and steering, as well as mobility. Specifically, we focus on how to construct interactive components that efficiently interact with target applications in distributed and dynamic heterogeneous systems. Toward these ends, we offer runtime customization for mirroring and mobility.

7 Conclusions and Future Work

This paper presents the concept of mirror objects and their Java-based implementation. Mirror objects support the development of efficient interactivity infrastructures for high performance applications, by enabling end users to view and control distributed and parallel high performance applications. Two implementations of mirror objects, one CORBA-compliant, the other using Java, interoperate in order to offer both high performance component monitoring and control and the ability to take advantage of the rich set of tools available in the Java domain. In addition, the JMOSS Java mirror objects described in detail in this paper offer features not easily realized in the CORBA domain, such as mobility, customizable migration and customizable mirroring. Customizable migration is useful to reduce communication bandwidth needs or delays, especially when migrating from stationary PCs to handheld devices. Customizable mirroring enables developers to control and reduce the amounts of information extracted from the high performance programs and/or physical devices remotely viewed and controlled.

Mirror objects may themselves be mirrored, thus enabling the construction of rich interactivity systems that both contain 'mirrors' of their target applications and offer new functionality to help end users understand and manipulate these complex targets. In particular, by supporting the use of mirror objects in web browsers it is easy to offer remote access and collaboration support via interfaces that range from high end machines to handheld or even simpler, web-enabled devices. Our intent is to support functionality like teachers in schools guiding students through a simulation running on a remote supercomputer, a corporate home office collaborating with field representatives equipped only with laptops and wireless links, or engineers operating in offices and on shopfloors.

Ongoing and future work with JMOSS includes optimizing its constructs for remote method invocation as well as experimentation with and automation of its customization features, specifically addressing mobility.

8 Acknowledgments

Ada Gavrilovska initiated the RTTB design workbench, and she provided its first MOSS-based implementation. Many researchers in the School of Mechanical Engineering at Georgia Tech have contributed to this effort, with particularly generous support from Jonathan F. Gerhard and Angran Xiao for their help in understanding the RTTB work and for their valuable comments on drafts of this paper.JMOSS is based on earlier work performed by Greg Eisenhauer and on previous research by Dong Zhou. We thank both for helping with this paper, as well. We also thank Neil Bright for his support in the software installations required for this effort.

References

[1] E. Akarsu, G. Fox, W. Furmanski, and T. Haupt. Webflow - high-level programming environment and visual authoring toolkit for high performance distributed computing. In *SC98: High Performance Networking and Computing*, December 1998.

[2] ANSYS,INC. *ansys Homepage*. `http://www.ansys.com/products/index.htm`.

[3] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski. Toward a common component architecture for high-performance scientific computing. In *Procedings of the 8th IEEE International Symposium on HIgh Performance Distributed Computation*, pages 115–124, August 1999.

[4] B. Bramley, K. Chiu, S. Diwan, D. Gannon, M. Govindaraju, N. Mukhi, B. Temlo, and M. M. Yechuri. A component based services architecture for building distributed applications. In *Proceedings of High Performance Distributed Computing (HPDC-2000)*, pages 51–59, August 2000.

[5] *Chemical Engineering Applications Technology.* `http://sharan.ncsa.uiuc.edu/chemengathome/home.page.docs/chemeng.html`.

[6] G. Eisenhauer. *An Object Infrastructure for High-Performance Interactive Applications.* PhD thesis, Georgia Institute of Technology, 1998.

[7] G. Eisenhauer, F. Bustamente, and K. Schwan. Event services for high performance computing. In *Proceedings of High Performance Distributed Computing (HPDC-2000)*, August 2000.

[8] G. Eisenhauer, B. Plale, and K. Schwan. Dataexchange: High performance communication in distributed laboratories. *Journal of Parallel Computing*, (24):1713–1733, 1998.

[9] G. Eisenhauer and K. Schwan. An object-based infrastructure for program monitoring and steering. In *Proceedings of the 2nd SIGMETRICS symposium on Parallel and Distributd Tools(SPDT'98)*, pages 10–20, August 1998.

[10] T. G. Gabrielle Allen, Werner Benger and etc. The cactus code: A program solving environment for the grid. In *Proceedings of High Performance Distributed Computing (HPDC-2000)*, pages 253–260, August 2000.

[11] J. Gerhard, , J. Allen, D. Rosen, and F. Mistree. A distributed product realization environment for design and manufacturing. submitted to ASME Computers and Information in Engineering Conference, 2000.

[12] J. Gerhard, S. Duncan, Y. Chen, J. Allen, D. Rosen, F. Mistree, and A. Dugenske. Towards a decison-based distributed product realization envoroment for engineering sysems. In *Proceedings ASME Computers in Engineering Conference (DETC/CIE-9085*, September 1999.

[13] W. Gu, G. Eisenhauer, and K. Schwan. Falcon: On-line monitoring and steering of parallel programs. *Concurrency: Practice and Experience*, 10(9):699–736, August 1998.

[14] J. K. Hollingsworth, B. P. Miller, M. J. R. Gonalves, O. Naim, Z. Xu, and L. Zheng. Mdl: A language and compiler eor dynamic program instrumentation. In *International Conference on Parallel Architectures and Compilation Techniques*, November 1997.

[15] J. Maassen, R. van Nieuwpoort, R. Veldema, H. Bal, and A. Plaat. An efficient implementation of java's remote method invocation. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'99)*, pages 173–182, May 1999.

[16] National Center for Supercomputing Applications and University of Illinois at Urbana-Champaign. *NCSA Habanero.* `http://havefun.ncsa.uiuc.edu/habanero/`.

[17] NCSA. *Portals to Scientific Computing.* `http://morefun.ncsa.uiuc.edu/cpa/`.

[18] NCSA Alliance. *Access grid.* `http://www-fp.mcs.anl.gov/fl/accessgrid`.

[19] NPACI. *Grid Portal Toolkit (GridPort).* `http://gridport.npaci.edu/`.

[20] G. M. Olson, D. E. Atkins, R. Clauer, T. A. Finholt , F. Jahanian, T. L. Killeen, A. Prakash and T. Weymouth. The upper atmospheric research colloboratory. In *Interactions*, 5(3):48–55, 1998.

[21] C. M. Pancerella, L. A. Rahn, and C. L. Yang. The diesel combustion collaboratory:combustion researchers collaborating over the internet. In *Proceedings of the ACM/IEEE SC99 Conference*, November 1999.

[22] Parametric Technology Corporation (PTC). *Pro/ENGINEER Homepage*. `http://www.ptc.com/products/proe/index.htm`.

[23] B. Parvin, J. Taylor, G. Cong, and M. O'Keefe. Deepview: A channel for distributed microscopy. In *Proceedings of the ACM/IEEE SC99 Conference*, November 1999.

[24] B. Plale, G. Eisenhauer, J. Heiner, V. Martin, K. Schwan, and J. Vetter. From interactive applications to distributed laboratories. *IEEE Concurrency*, 6(2):78–90, April-June 1998.

[25] D. Rosen, Y. Chen, J. Gerhard, J. Allen, and F. Mistree. Design decison templates and their implementation for distributed design and fabrication. submitted to ASME CIE, DAC, DFM Conference, Joint Sessions on SFF, 2000.

[26] D. W. Rosen. Progress towards a distributed product realization studio: The rapid tooling testbed. In *3rd IFIP WG 5.2, Proceedings of Workshop on Knowledge Intensive Cad (KIC-3)*, December 1998.

[27] D. Silva and K. Schwan. Ctk: Configurable object abstractions for multiprocessors. to appear in IEEE Transactions Software Engineering, also see GIT-CC-97-03.

[28] M. Straer, J. Baumann, and F. Hohl. Mole: a java based mobile agent system. In *M. Muehlhuser: (ed.), Special Issues in Object Oriented Programming*, pages 301–308, 1997.

[29] J. Vetter and K. Schwan. High performance computational steering of physical simulations. In *IEEE International Parallel Processing Symposium (IPPS)*, April 1997.

[30] D. Zhou, K. Schwan, G. Eisenhauer, and Y. Chen. Supporting distributed high performance application with java event channels. to appear in the 2001 International Parallel and Distributed Proceeding Symposium(PDPS), April 2001.

A Sample Wrapper Object

The following is a sample code segment for the RTTB design workbench, including the mirror interface and the `State` class defined by the user, and the wrapper class generated by *jmossw*. The original Java class `Experiment` interacts with the target application via a MOSS mirror not shown in this code fragment:

```
/* user defined mirror interface */
  public interface RemoteExpInterface extends java.rmi.Remote {
   static final String appName="RTTB";
  State  state;
        ...
   int runProe(String trailFile,String proeFile)throws RemoteException;
   int runAnsys(String proeFile,String ansysFile)throws RemoteException;
   int runFdmaxi(String ansysFile) throws RemoteException;
   int start(int N2, float Fw, int Pd, float Load, String
   trailFile) throws RemoteException;
   int pause( ) throws RemoteException;
   int stop( ) throws RemoteException;
   int resume( ) throws RemoteException;
}

/* user defined state class */
  class State {
   int status;
   float finalResult;
   ...
}
```

```
/*wrapper class generated by jmossw */
class RemoteExperiment extends UnicastRemoteObject
       implements RemoteExpInterface, jecho.PushSupplier{
   protected Experiment appObject;
  PushSupplierHandle tpsh;

   //associate the application object with wrapper object
   RemoteExperiment(Experiment app) throws RemoteException {
       appObject = app;
       JMOSS_init(RemoteExpInterface.appName);
       appObject.wrapperObject = this;
   }

   // initialize event channel and RMI call
   void JMOSS_init(String appName) {
    try {
     tpsh=new PushSupplierHandle (this,"java.lang.Object");
     String chanName = "JMOSS-"+appName;
     String bindName = "//localhost"+"/"+"JMOSS-"+appName;
     tpsh.connectTo (new EventChannel(chanName,
                                      "java.lang.Object")))
     RemoteExpeInterface exp = new RemoteExperiment();
     Naming.rebind(bindName, this);
     }catch (Exception e) { e.printStackTrace();}
     state = new State( );

   }

   // attribute access methods
   void  experiment_Set_status(int command) throws RemoteException {
        state.status = command;
        //user's code to change application object's state
        tpsh.send(state,true);
   }
   int experiment_Get_status( ) throws RemoteException {
       return State.status;}
          ...

   // methods defined in mirror interface
   int start(int N2, float Fw, int Pd,float Load, String trailFile)
               throws RemoteException {/*user's code */}
                    ...
   int pause( ) throws RemoteException {/*user's code;*/}
   int resume( )throws RemoteException {/*user's code;*/}
}
```

`State` and `RemoteExpInterface` are the user-defined 'state' class and mirror interface, respectively. `RemoteExperiment` is the wrapper class generated by *jmossw*. The application-level 'object' described by class `Experiment` is not shown here.`State` defines the state contained in the mirror object. In the trivial case, this state exactly corresponds to the state of the application object being mirrored. `RemoteExpInterface` defines methods that mirror objects can call to steer the application. `RemoteExperiment` implements the mirror interface. Its constructor assigns the wrapped application object to variable appObject, creates event channels, registers the remote object, and notifies the application of the wrapper object. `Wrapper` is an event provider from the mirror object's point of view. The events it provides as a JECho `PushSupplier` are the state changes occurring in the instrumented application object that are of interest to the Java mirror.

Following these definitions, the `JMOSS_init` called by the constructor initializes the RMI call mechanism, creates the event channel for transporting monitoring information from the application to the mirror object

(JMOSS currently uses one event channel per monitored application object), binds the event channel to the mirror object, and finally, creates the mirror state (see the `new State` call). As stated earlier, accesses to state attributes occur only via get/set methods that are generated by the wrapper generator. The methods defined on this particular object get and set certain experiment attributes. An `attribute set` method call on a wrapper object results in an update of the wrapper and application's state that is also propagated to the mirror objects (see the `tpsh.send` call in the code). This call offers an option as to whether such updates should be done asynchronously or synchronously ('true' means 'synchronous' and is the default option). An `attribute get` call returns the desired application state, not by actually calling the application object, but by reading and returning the wrapper's state. Recall that its state updates are performed automatically, via events pushed by the application-level event provider.

The final lines of code in this example involve running, pausing, and stopping the actual application components remotely controlled via the RTTB design workbench. These components include codes like ProE[22], Ansys[2], and experimental packages like FdMaxi. Such creation and control is done via user-provided operations defined on application components.

An additional variable, `wrapperObject`, which refers to the wrapper object, should be added to the application class definiation. Wherever user-defined state is changed in the application, a method call statement which invokes the wrapper's `set` method should be inserted. This ensures that the wrapper's state is changed and that these state changes are propagated to the appropriate mirror objects.

B Sample Mirror Object

The example shown here is a sample JMOSS mirror object `MirrorExperiment` used in RTTB runs, which is generated by mirror generator*jmossm*.

`MirrorExperiment`'s initialization method subscribes to the event channel and then acquires a remote reference of wrapper object. `MirrorExperiment` has all methods defined in mirror interface. The implementations of these methods are remote method invocation on the wrapper object. User should implement push( ) method, which handles state updates.

```
public class MirrorExperiment implements jecho.PushConsumer{
   PushConsumerHandle dpch;
  RemoteExpInterface exp;
   State  state;

  MirrorExperiment(appName, remotehost) {
        JMOSS_init(appName,remotehost);
  }

  JMOSS_init(String appName,Stirng remotehost) {
    dpch = new PushConsumerHandle(this, new PolicyObject( ));
    try {
       String chanName = "JMOSS-"+appName;
       dpch.connectTo (EventChannel.open (chanName),-1))
       String bindName = "//remotehost"+"/"+"JMOSS-"+appName;
       exp = (RemoteExpInterface) Naming.lookup(bindName);
                  ...
     } catch (Exception e) {
       System.err.println("Monitor exception:"+e.getMessage());
       e.printStackTrace();
  }

  int runProe(String trailFile,String proeFile){
      exp.runProe(trailFile,proeFile);}
```

```
      ...

     ...
  int pause( ) throws RemoteException {
       exp.pause( );
       ...
   //user's code to process state update
   public void push (Object obj) {
       float minStress = 3.0;
       State state = (State)obj;
       System.out.println("Current status:" + state.status);
       ...
       if(state.status == DONE || state.result >= minStress)
         System.out.println("OK");
       ...
    }
}
```