Java Mirrors: Building Blocks for Remote Interaction

Yuan Chen, Karsten Schwan
College of Computing
Georgia Institute of Technology
{yuanchen,schwan}@cc.gatech.edu

David W. Rosen
Mechanical Engineering
Georgia Institute of Technology
david.rosen@me.gatech.edu

Abstract

Java mirror objects are building blocks for integrating web-based with high performance applications. This paper demonstrates the utility of mirror objects for construction of remote 'portals' via which scientists or engineers can view the results of ongoing large-scale simulations or physical processes, and even control them, from a wide variety of remote platforms. A sample portal constructed with mirror objects is the RTTB design workbench, which controls components of the RTTB rapid tooling and prototyping testbed. Here, mirror objects continuously mirror the states of potentially remote RTTB software or hardware entities, thereby enabling remote inspection. Remote control is enabled by automatically propagating the operations performed on mirrors to these entities. As a result, mirrors effectively virtualize remote entities. In addition, the JMOSS mirror object implementation described and evaluated in this paper exhibits functionality that is important for mobile portals, including their interoperation with a high performance, CORBA-based implementation of mirror objects and their customization of mirroring so that workbench users can continue their PC-based online interactions via handheld devices used on the shopfloor.

1. Introduction

The Internet has created new opportunities for remote interaction and collaboration. Scientists and engineers working in geographically different locations remotely visualize the results of their large-scale simulations[14], and their models use real-time information captured by remote instruments[7]. In addition, real-time collaboration tools[4] permit evaluation and discussion of results and insights with remote colleagues [10]. Finally, modern portal technologies[1, 11] and their underlying publish/subscribe communication infrastructures[5, 18] not only enable the real-time viewing and inspection of the results of remote simulations and/or instruments, but also their online steering and control[14, 17].

Portals and Virtual Workbenches for Scientists and Engineers. The purpose of *portals* is to give users remote access to resources ranging from digital library data[11], to remote instruments[7], to information produced by running simulations[14] or even captured on the shopfloor. This paper uses the term *virtual workbench* rather than portal, in order to indicate the importance we place on users' abilities to both access and manipulate remote entities via these interfaces. As an example, we present an interactive 'Design Workbench' for the Rapid Tooling Testbed(RTTB)[16] and used by researchers in Mechanical Engineering at Georgia Tech. The RTTB permits engineers to rapidly design, simulate, and prototype new mechanical parts. The 'Design Workbench' presents to such users a virtualization of the RTTB, which enables 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 by the RTTB. 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, as with most high performance computations[10] or Internet resources, 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. [1]

Mirror Objects – Building Blocks for Distributed Workbenches and Portals. Mirror objects are the key build-

[1] Other research groups have formulated similar characterizations for scientific or engineering 'workbenches' or 'portals' [11, 2].

ing blocks used to realize the RTTB workbench, because they 'mirror' those behaviors of the target application being viewed or controlled that are important to end users. Specifically, a mirror object is a CORBA- or Java-based representation of an application component that may itself not be structured as an object.The idea is to 'cast' into the structured forms of objects components of an engineering or scientific application written in Fortran or C. This involves instrumenting the components such that (1) any updates of their states important to remote users trigger consequent updates of the mirror objects' states, and (2) when invoked, operations exported via the mirror objects' class interfaces trigger updates on the corresponding application components. In this fashion, operations performed on the remote workbench's mirrors are turned into operations performed on the target application, thus making mirror objects into virtualizations of applications components.

JMOSS Java Mirrors. The use of mirrors across the Internet and from mobile devices requires novel capabilities from the Java-based JMOSS mirrors described in this paper:

Mobility – JMOSS mirrors can be migrated at any time, without loss of messages in transit. This facilitates the implementation of functionality in which an end user moves his interactions from a desktop to a mobile device and back to the desktop, thus enabling him to freely move between lab and shopfloor, for instance.

Customized Mirroring and Migration – JMOSS mirrors can be customized, and mirror migration specialized, the former permitting users to have differing views of an object's state, the latter permitting mirror state to be changed during migration. Customized migration is important in mobile systems,where the object state being migrated must be adjusted to the differing capabilities of remote devices like desktops vs. handhelds. Customized mirroring is critical when insufficient communication bandwidths preclude the transfer of entire mirror states. Customization permits Java-based mirrors to 'keep up' with the high performance RTTB simulation by only transmitting and converting the data currently most important to an end user.

Interoperability with non-Java systems – JMOSS mirrors interoperate with the MOSS CORBA-based realization of mirror objects developed in our previous work and shown useful for the runtime monitoring and steering of high performance applications[6]. They also interoperate with existing engineering or scientific applications, by automatic conversion of their typed output data to Java objects (and vice versa). Our results show that the conversion of typical workbench data from its native form to Java objects typically adds less than 15% to data transfer costs.

Technical Contributions. Our previous work has demonstrated the utility of the MOSS CORBA-based realization of mirror objects for the runtime monitoring and steering of high performance applications[6].This paper focuses on the properties of mirrors required for their use across the Internet and in mobile systems: (1) the ability to migrate mirrors, (2) the customization of mirroring and of mirror migration, and (3) interoperability of Java- with non-Java-based mirrors. Toward these ends, this paper employs both CORBA- and Java-based mirror objects for remote program inspection and control,where the JMOSS Java mirrors are used to extend mirroring into Internet-connected and mobile devices.

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 application's instrumentation. Since automatic instrumentation requires tools not freely available, instrumentation is performed manually for the RTTB workbench.[2] Mirror objects are 'linked' to the target applications via events updating their states in response to application-level state changes. Such monitoring events are generated for all state changes in instrumented application-level objects. Events are asynchronous, in that application-level state changes are not delayed until mirror-level state changes have been completed. The motivation is to avoid imposing unnecessary overheads on the high performance applications monitored and controlled by mirror objects.

Since mirror objects are intended to be faithful virtualizations of application components, methods executed on mirrors that change their internal states reflect such changes on application-level components. This is done via synchronous remote method invocations that trigger 'steering' actions on application components.

Mirror objects also contain the additional methods and/or derived state implied by the roles they play in virtual workbenches. They may implement certain state transformation of display methods, for instance, so that they render transformed and not raw, unintelligible application state[14, 18].

Application programs may be mirrored many times and in many places, as per their instrumentation, and a single application 'object' may be associated with any number of mirror objects. In both cases, by default, all mirrors observe the same application-level changes, and an operation executed on a mirror is synchronously reflected to all application-level objects being mirrored. In addition, mirror objects themselves may be further mirrored, thereby creating hierarchies of mirrors. Thus, inherent in mirror objects is basic support for multiple observers to view the same

[2] Better instrumentation methods include the source code annotations[6, 8] used in our own past research, compiler and linker support, or runtime binary editing[9].

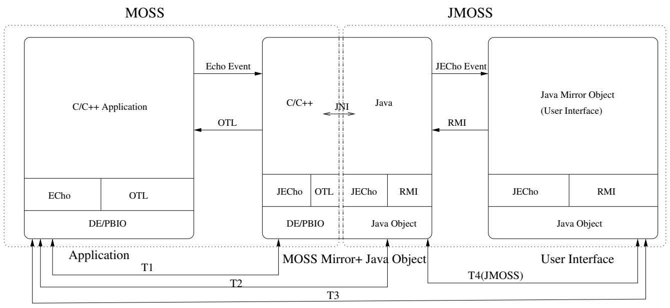

Figure 1. Mirror Object Architecture and Experiment Configuration

data, each in ways customized to their individual needs.

Implementation. The ECho and JECho publish/subscribe event infrastructures[5, 18] implement the communications within mirror object structures, as depicted in Figure 1. Specifically, each mirror object subscribes to the event channels to which the application-level monitoring events they desire are sent. Thus, any one mirror object can receive events emanating from any number of application components and/or remote objects, for instance to track the durations of certain application-level actions. Conversely, to propagate to application components the state changes resulting from operations on mirror objects, mirror objects use object invocation layers accessible in ECho and JECho applications, which is the Object Transport Layer (OTL) in ECho and RMI in JECho, both using IDL interface descriptions.

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. Finally, it is the Java mirror object that either acts as or interacts with some wired or mobile Java-based user interface.

3. JMOSS Implementation

3.1. Basic JMOSS Functionality

Interoperation of MOSS and JMOSS Objects. MOSS and JMOSS objects interact via the Java native interface (JNI). Specifically, a MOSS mirror produces events described in some standard native form. This form is translated into a Java object by JMOSS's Java/native translation library. Conversely, a Java mirror object interacts with a MOSS object by calling native procedures in the MOSS library, again using the translation library to create native events from Java objects.

Java Objects and JMOSS Mirrors. JMOSS may be used to mirror Java objects or MOSS' CORBA-based objects. The following process is used to 'ready' an existing Java object for remote monitoring and steering and then create its mirror object. First, the Java object is 'wrapped'. Each such wrapper controls access to the object's internal state. Such state is read by calling the wrapper's 'get attribute' methods, and it is updated with 'set attribute' methods. Second, the wrapper propagates state updates to all JMOSS mirrors associated with this object, by creating appropriate event channels, subscribing to such channels as a provider, and publishing state changes to all JMOSS mirrors that have subscribed to the channel. Third, all changes to the Java object's state are mediated by the wrapper, thereby ensuring that its state updates can be propagated to its dependent mirrors.

JMOSS provides the *jmossw* wrapper generator tool to assist users in the creation of wrappers for existing Java object. JMOSS mirror objects are generated using the *jmossm* tool. Once created, they receive state updates by subscribing to appropriate JECho channels created by their wrappers. Steering operations are accomplished by invoking the wrapper object's methods using RMI. Further details of the JMOSS implementation,including sample code segments for the RTTB design workbench appear in a technical report[3].

3.2. Advanced JMOSS Functionality

Customized Migration. JMOSS mirror objects can migrate while in use and ensure that additional updates received during migration are not lost. *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. For (1) and (2), users may provide their own serialization and deserialization, by implementing the jecho.JEChoObject interface. This interface is similar to the java.io.Externizable interface, except that it uses JECho's optimized and customized object stream. The user can declare only certain object fields to be serialized and other fields to be ignored. At the destination, during deserialization, the user-provided procedures may recompute the values of fields that were not migrated. For (3), the 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).

Customized Mirroring. *Customized mirroring* controls how state updates on mirror objects are performed. This is in contrast to previously explained customized migration, which permits changes in the amounts of state migrated and the ways in which state is initialized and used at the target. Customized mirroring is important because mirrors are updated continuously, in accordance with changes in the target application component being mirrored. It permits the developer, for instance, to 'move' at runtime an averaging computation being performed on two values from the mirror to the object being mirrored, thereby reducing communication

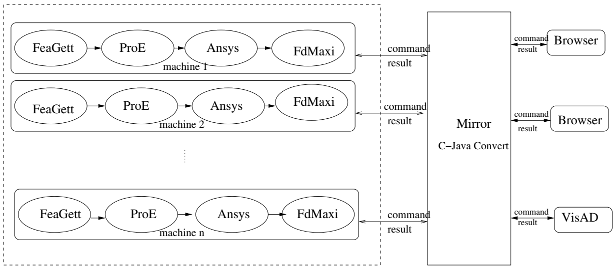

Figure 2. RTTB workbench

overheads at the cost of only slightly increased perturbation in the object being mirrored. This is particularly important for mobile mirrors, where large-scale data transfers should be prevented from crossing wireless links.

Customized mirroring is implemented using lower level support for handler migration offered by the JECho event transport facility (see [18]). 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.

Concurrency and Consistency Control. There is also concurrency and consistency control in JMOSS.

4. The RTTB Design Workbench

The workbench targets the RTTB(Rapid Tooling Testbed), which intended to be a distributed computing environment to support product design, prototyping, and manufacturing[15]. The engineers using it are experimenting with different design processes and different sets of tasks, personnel, vendors, software, and equipment. [3]

The experiments with the RTTB workbench conducted in this paper use four software packages to iteratively design and analyze a new part. They are the experiment designing and parsing package FeaGett, the 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. The others are commercially available codes.

A typical sample procedure involves generating a trail 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 objects that contain all relevant attributes and status information. 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 distributed computations, not only final results 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, using local or browser-based interfaces that employ mirror objects. Intermediate results are typically quite large. At any one time, 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.

The experiment executes multiple simulations and analyses in parallel. Simulations run on parallel machines and on workstations connected via 100MB Ethernet. Researchers use both networked, lab-resident workstations and wireless Linux-based iPAQ pocket PCs to control and monitor ongoing experiments.

Figure 2 depicts the RTTB workbench architecture built using MOSS and JMOSS, including the parallel nature of ongoing simulations, the mirroring that targets multiple end users and user interfaces, and the online control exerted via those interfaces.

5. Performance Evaluation

5.1. Basic Benchmarks

Figure 1 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. 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. The times required to complete a roundtrip through a MOSS mirror and ending up in a Java object using the JNI interface are depicted in Table 1. This represents the minimum delay experienced 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 this 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.

Results depicted in Table 1 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 ar-

[3]As part of the RTTB effort, multiple computing infrastructures have provided communication, information sharing, work-flow, and distributed computation capabilities. The design workbench described in this paper constitutes one such infrastructure.

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

Table 1. Elapsed real-time for MOSS(T1)and MOSS+Conversion(T2)

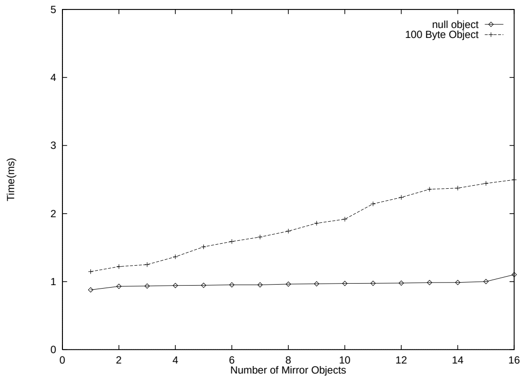

Figure 3. Elapsed Real-time for JMOSS Using Different Number of Mirror Objects

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

Table 2. Migration Time

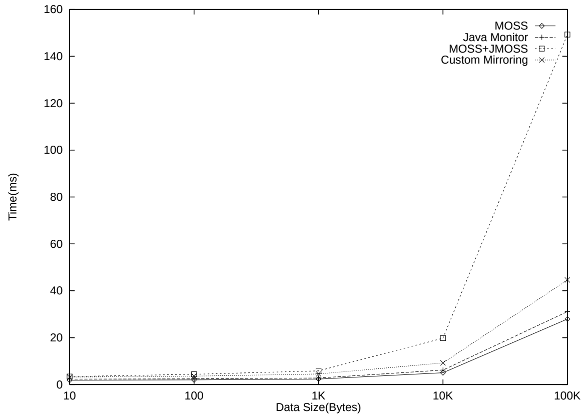

Figure 4. Elapsed real-time of MOSS(T1), MOSS+Conversion(T2), JMOSS(T3) without Custom Mirroring and JMOSS(T3) with Custom Mirroring

ray 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%.

Scalability of Mirror Objects. To support collaboration, multiple mirror objects may be created for a single application component. Figure 3 depicts JMOSS round trip costs when varying the numbers of mirror objects. Results show that steering latency does not vary significantly for up to 16 mirror objects. Measurements are performed on a cluster of 300MHz dual Pentium II Linux PC connected with 100Mbps fast Ethernet. Each mirror object is located on a different machine.

Comparison to Java Sockets. Results detailed in [3] demonstrate that JMOSS round trip times are comparable to Java socket communication delays, including for large data transfers.

5.2. Advanced JMOSS Functionality

In the following measurements, an application object runs on a workstation, and a mirror object resides on an iPAQ H3650 running the Linux operating system. 802.11b wireless LAN communication devices connect both. (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).

Customized Migration. Table 2 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 via 100MB Ethernet. The second setup uses one workstation and one iPAQ H3650 handheld device connected via the 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 costs 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 customized mirroring, using end-to-end measurements between a JMOSS Java mirror object and a native program. The times shown are the delays of MOSS, MOSS+conversion and JMOSS (the sums of MOSS, data conversion, and JMOSS delays) for different amounts of data. These results indicate that JMOSS mirror objects are somewhat impractical for mirroring large amounts of state (JMOSS latency is more than 5 times larger than MOSS for 100KB data), thus arguing for customizing mirroring to suppress undesired state information. The simple demonstration of customized mirroring used in Figure 4 is one in

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

Table 3. Elapsed Real-time of JMOSS on Wired vs. Wireless networks

which only 10% of total state is mirrored from the MOSS to the JMOSS mirror. This shows that JMOSS response times after customized mirroring are comparable to MOSS, resulting 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 be intermittent. Table 3 lists the basic costs experienced by JMOSS objects when using a wireless network. In comparison to the latencies experienced over a wired network, results show that the latency increases by factor of more than 70. This is due to the limited bandwidth available in our wireless domain, typically less than 300kbs. In environments like these, functionalities like customized mirroring and migration are critical for achieving what appears to end users real-time, interactive mirroring.

6. Conclusions and Related Work

This paper presents the concept of mirror objects and their use for construction of efficient remote portals or workbenches. To address the potentially large data transfers required from applications to portals in the domain of high performance computing, two implementations of mirror objects, one CORBA-compliant, the other using Java, interoperate in order to offer both high performance and flexible component monitoring and control. Furthermore, JMOSS Java mirror objects have properties that are important to their use with portals across the Internet and/or in ubiquitous systems, including mobility, customizable migration, and customizable mirroring.

Mirror objects may themselves be mirrored, thus enabling the construction of rich workbenches that contain 'mirrors' of their target applications and also offer new functionality. Furthermore, by supporting the use of mirror objects in web browsers, remote access and collaboration support can use interfaces that range from high end machines to handheld or even simpler web-enabled devices.

Related Work. The Diesel Combustion Collaboratory (DCC)[12] was a pilot project to develop and deploy collaborative technologies to combustion researchers distributed throughout the DOE national laboratories, academia, and industry. 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[13] 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[5, 6]. JECho and thus, JMOSS also demonstrate improved performance for state mirroring compared to other Java-implemented event systems. More comparisons of related work appear in a technical report[3].

References

[1] 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 *Proceedings of the 8th IEEE International Symposium on HIgh Performance Distributed Computation*, pages 115–124, August 1999.

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

[3] Y. Chen and K. Schwan. Java mirrors: Building blocks for interacting with high performance applications. Technical report, College of Computing, Georgia Institute of Technology, 2001. Submitted for review. `http://www.cc.gatech.edu/ yuanchen/tech.ps`.

[4] R. C. G. Daniel A. Reed and C. E. Catlett. Distributed data and immersive collaboration. *Communications of the ACM*, 40(11):38–49, November 1997.

[5] G. Eisenhauer, F. Bustamente, and K. Schwan. Event services for high performance computing. In *Proceedings of High Performance Distributed Computing-9(HPDC-9)*, August 2000.

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

[7] T. A. Finholt and G. M. Olson. From laboratories to collaboratories: A new organizational form for scientific collaboration. *Psychological Science*, 9(1):28–36, January 1997.

[8] 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.

[9] 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.

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

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

[12] 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.

[13] B. Parvin, J. Taylor, G. Cong, and M. O'Keefe. Deepview: A channel for distributed microscopy. In *Proceedings of the Supercomputing 1999(SC99)*, November 1999.

[14] 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.

[15] D. Rosen, Y. Chen, J. Gerhard, J. Allen, and F. Mistree. Design decision templates and their implementation for distributed design and fabrication. In *ASME DETC 2000 Conference,DAC-14293*, 2000.

[16] 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.

[17] J. S. Vetter and D. A. Reed. Real-time performance monitoring, adaptive control, and interactive steering of computational grids. *The International Journal of High Performance Computing Applications*, 14(4):357–366, 2000.

[18] D. Zhou, K. Schwan, G. Eisenhauer, and Y. Chen. Supporting distributed high performance application with java event channels. In *Proceedings of the 2001 International Parallel and Distributed Processing Symposium (IPDPS 2001)*, April 2001.