Computational Study Support for

Problem-Solving Environments *

Siu-Man Yau; Akash Garg; Jeremy Archuleta;
Eitan Grinspun; Vijay Karamcheti; Steven G. Parker; Denis Zorin

* Courant Institute of Mathematical Sciences, New York University
† Department of Computer Science, Columbia University
‡ Department of Computer Science, University of Utah

{smyau,vijayk,dzorin}@cs.nyu.edu,
{akash,eitan}@cs.columbia.edu,
{jsarch,sparker}@cs.utah.edu

Abstract

This paper describes and evaluates mechanisms for supporting interactivity in the SimX parallel system for multi-experiment computational studies. SimX is built on top of the SCIRun modular problem-solving environment and enables users to interact at the level of entire computational studies, changing their inputs based on results from already run experiments.

To support this usage model while also maximizing the size of studies that can be performed at interactive rates, SimX relies on three groups of mechanisms: (1) study-level visualization and steering mechanisms; (2) extensive reuse of results to improve both study- and experiment-level responsiveness; and (3) system-level mechanisms for checkpointing and a shared object layer, which enables the reuse. Evaluation using a SCIRun defibrillator device simulation code on a parallel cluster highlight the advantages of these mechanisms, demonstrating that they do in fact enable fairly sophisticated studies to be conducted at or close to interactive rates.

1 Introduction

SimX is a parallel software system for conducting interactive multi-experiment computational studies. Its design was motivated by the recognition that computer simulation has become an integral part of the scientific method,
often delivering deeper insights into complex physical processes than possible using only the traditional dyad of theory and experiment. Computer simulation manifests itself in the scientific exploration process in the form of computational studies built out of multiple computational experiments corresponding to individual runs of simulation software. Examples of such studies range from exploration of design spaces in engineering to molecular simulations for drug design. Driven by the availability of higher-performance computational resources, the number of experiments involved in computational studies has increased dramatically, and their structure has become more complex.

However, what has not fundamentally changed is how such studies are conducted. Typically, the scientist (1) makes some a priori decisions about simulation parameters; (2) runs the simulation, or a batch of simulations; and (3) analyzes the output. These steps are repeated as necessary; the organization of the study is left to the scientist or domain application developers. While this is a reasonable pattern for studies involving small numbers of experiments, for hundreds and thousands of experiments, the need to set up individual experiments manually or using simple predetermined parameter variation patterns becomes a severe limitation. For efficient exploration, the scientist needs high-level tools for manipulating collections of simulations, and the ability to adjust the parameter space traversal pattern continuously, based on partial results aggregated from large numbers of running experiments.

The SimX system aims to support intuitive, high-level management of computational studies by permitting users to interact at the level of aggregate studies (instead of individual experiments), by providing continuous feedback about running simulations, and by dynamically adapting allocation of system resources to reflect changing priorities set by the user. To achieve these goals, SimX relies on a more permeable interface between parallel system software and numerical simulation codes than is assumed by state-of-the-art middleware in grid computing [18, 3, 11, 16] and problem solving environments [19, 13, 10] that target related problems. These interfaces help in two primary ways:

- they enable SimX to gather knowledge about how individual experiments are contributing to progress towards overall study goals; and
- they provide information about the internal state and resource requirements of individual experiments to enable substantially more efficient mapping of these experiments to computational resources.

A prototype standalone implementation of the SimX system was described in an earlier paper [26], along with a simplified bridge design study highlighting the kinds of system-level optimizations SimX is capable of performing, and their impact on the efficiency of adaptive design space exploration performed by the system. The original system did not support such essential features as interactive user modification of performance measures, areas of interest and admissible ranges for performance measures. Even more importantly, the stand-alone nature of the system made it more difficult to integrate with existing applications.

This paper describes a new incarnation of the SimX system based on the University of Utah’s SCIRun problem solving environment [19]. SCIRun is a modular, component-based system for interactive scientific computing applica-
tions, which permits the expression of a simulation experiment along with its steering and visualization interfaces as a dataflow net. SCIRun provides the foundation for the CCA-compliant [2] SCIRun2 framework [27]. Implementation of the SimX system within SCIRun offers several advantages:

1. It permits natural component-level integration of the various SimX runtime modules with existing simulation functionality already expressed in component form (a monolithic simulation code would have required extensive rewrites to provide the permeable interface SimX relies upon);

2. It enables us to leverage existing scientific computing software, especially tools supporting user interaction and visualization, both in SCIRun/SCIRun2 proper and in other toolkits to which SCIRun2 provides an interface, including CCA;

3. It ensures access to our techniques through a well-established framework, which we expect to be increasingly common in scientific computing applications, thus increasing the potential for practical impact.

We have experimented with the new SimX implementation using a computational study built around an existing SCIRun-based defibrillator design simulation code (DefibSim) [22]. Only minimal changes were necessary to the simulation (specifically, separation of the visualization and simulation data nets and addition of modules providing communication). This allowed us to explore quantitatively the advantages of reuse and adaptive computational study management in a realistic setting, and exposed several avenues for future development of the SimX system.

The rest of this paper is organized as follows. Section 2 provides the required background for the paper, discussing in turn, related efforts, the SCIRun environment, the motivating DefibSim application, and the overall SimX architecture and its standalone implementation. Sections 3 and 3.2 describes the SimX implementation within SCIRun and our experiences in creating an interactive steering environment for the defibrillator study using SimX and SCIRun. We evaluate the system in Section 4 and conclude by identifying several next steps for the SimX system in Section 5.

2 Related Work and Background

2.1 Related Work

The overall goals of the SimX system overlap with previous efforts in computational steering infrastructures and parameter sweep applications.

Computational steering infrastructures provide system-level support to permit users to continually inspecting the state of running computations, and “steer” the computation as appropriate (e.g., by varying internal parameters of the computation). Some important examples include Falcon [12], CUMULVS [10], UINTAH [8], DISCOVER [17], CSE [24], RealityGrid [4], WEDS [7], and gViz [25]. The system we have developed is based on SCIRun [20], a computational steering infrastructure and problem-solving environment developed at the University of Utah.
Our work shares with these systems the ideas of standardized component architectures, enabling application developers to mix-and-match compute, steering, and visualization modules and reuse generic modules across multiple application domains (e.g., SCIRun/BioPSE’s ShowField module[14]); dynamic (dis)connection of interaction and visualization modules [15]; and support for a low latency visualization/steering interaction cycle [21]. However, in marked contrast to most of these systems, which have focused on steering individual (or a small number of) experiments, our work looks into steering computational studies involving a multitude of experiments. The latter necessitates both a higher level of user interaction than in the case of individual experiments, as well as more sophisticated resource management policies.

The other body of related work arises in the context of grid computing infrastructures such as APST [5], Nimrod [1], Condor [23], Globus [3], Netsolve [6], and Virtual Instruments [9], which provide support for the scheduling of parameter sweep applications, where the same application is run with a change in parameter values across distributed resources. With few exceptions, these schedulers assume that the simulations are independent and hence can be run in batch mode. A key insight behind the SimX architecture is that the simulations are in fact dependent, and exploiting these dependencies can yield substantial study runtime improvements.

Moreover, other than the exceptions listed below, parameter sweep tools support only limited forms of user interaction. Condor DAGman permits specification of a workflow involving several related parameter sweep jobs, Nimrod/O [1] supports applications where the user’s interest is in identifying the point in parameter space that maximizes a particular objective function, but cannot handle dynamically changing “soft” preferences. Virtual Instruments (VI) [9] implements priority-based resource allocation for the application’s tasks, but does not support reuse or high-level user interaction.

### 2.2 SCIRun Application Structure

SimX focuses on the unique support required to support computational studies involving hundreds to thousands of individual experiments. This support can be layered on top of traditional computational steering infrastructures, as we describe in this paper in the context of the SCIRun environment.

SCIRun applications are composed out of modules connected together via datalinks from the output data port of a module to the input port of another [19]. The resulting directed acyclic graph is called a dataflow net, a reference to SCIRun’s dataflow model of computation.

The SCIRun runtime associates each module with its own thread, which executes the module’s code. The code blocks until the data in the module’s input ports are available, does some calculation, and then sends data to the module’s output port. The SCIRun environment flows this data to the input ports of other modules, and triggers their execution. Thus, executing a module can cause all the modules reachable from to it to execute in the DAG’s partial...
order, as the data is sent down the DAG.

Each module can also optionally be associated with a user interface (UI) module, which is executed whenever the user clicks on a module. The UIs are the way a user can interactively steer a SCIRun application, by altering the parameters used in the code associated with the application module and viewing any information the latter provides.

### 2.3 SimX Architecture

A SimX computational study involves the systematic exploration of a design or parameter space of computational experiments so as to identify a desired target set. The latter is usually indirectly specified, in terms of the design space points where the experiment outputs defining the performance or observation space, satisfy certain constraints. An example of a computational study used in [26] is a simple 2d bridge design problem: an elastically deforming bridge has four supports, two of which are fixed at the endpoints and the location of the other two are chosen by the user. The objective was to find a set of bridge designs, which best traded off among two performance measures: the cost of bridge construction and the maximal deformation of the bridge. Formally, this objective translates to identifying the Pareto optimal points, the target set of parameter points such that there is no other point that can improve upon all dimensions of the performance metric.

SimX permitted this study to be managed at a high level: the user provided (1) the individual computational simulations, capable of computing deformation of the bridge (modeled as a one-dimensional rod elastically deforming in two dimensions) given specific values for the parameters and a predefined cost function; (2) specifications of the parameter and performance space domains; and (3) requirements on the target set of interest. The underlying system automatically made decisions about which parameter points to simulate, and in what order, so as to provide increasingly refined estimates of the Pareto frontier.

**SimX Runtime Modules.** The SimX system supports such computational studies using the high-level architecture shown in Figure 1. The core functionality of the system is realized by two modules, the active sampler and the resource allocator. A shared object space layer provides a machine-wide repository of shared state, including both simulation checkpoints and different meta-information about the ongoing study.

The active sampler converts user specifications of parameter and observation space domains, the target set, and priority/time target/precision functions into a collection of sample points in the parameter domain for which simulations need to be run. Whenever new user input arrives (communicated to the active sampler by the user interface modules), the sample set is adjusted. The active sampler occupies an intermediate position between system and application software. Adding domain knowledge to the sampler is likely to enhance its performance, but narrow the applicability of the system; making the sampler completely application-independent may result in suboptimal sampling strategies in important cases.
The resource allocator manages the pool of simulations of the computational study. It receives its directives from the active sampler module via a task list, and responds by starting new simulations and modifying the parameters of or terminating active ones. The goal of the resource allocator is to optimize completion time for these simulations.

**Standalone SimX.** The standalone implementation of the SimX architecture described in [26] relied on two types of communicating processes: *managers* and *simulation containers*. Simulation containers constitute the worker pool to which the manager farms off individual simulations. These processes communicate via a generic satellite interface; in the standalone implementation, the processes are just standard UNIX processes, and the satellite interface is implemented using TCP socket calls.

In addition to these explicit interactions, the processes also implicitly communicate using the shared object space layer. Simulations running within the simulation containers write checkpoints and results to this layer, while manager processes optionally write meta-information about the study; this information is read by other simulations and the manager. The dominant usage pattern indexes the information using the parameter space coordinates, so our implementation of the layer, SISOL, provides a spatially-indexed interface: objects are associated with spatial coordinates, and can be retrieved using neighborhood queries.

### 2.4 Defibrillator Design Computational Study

To describe the SimX system-level support for interactive computational studies, we use as a running example a computational study looking at defibrillator device design. This study also serves as the basis for the performance evaluation of the SimX system, which we describe in Section 4.

The underlying simulation code for the defibrillator design study is an existing SCIRun application net, Defib-Sim [22], which takes as its inputs a mesh representing the conductivity of the human torso, the positions of two electrodes, and the potential difference between the electrodes, and calculates as its output the electric potential inside the torso mesh.

The problem is governed by the Poisson equation relating the local conductivity tensor, the voltage over the domain, and the current source. The discrete form of the equation approximates the divergence of the electric field with the stiffness matrix $A$ and the voltages at the nodes with the vector $\Phi$. For a zero current source, they yield the system: $A\Phi = 0$. The electrodes at the front and back electrodes are modelled as Dirichlet Boundary conditions, and are incorporated into the equation by eliminating from $\Phi$ the nodes with known potentials. The simulation code solves the system $A'\Phi' = b$, where $\Phi'$ is the set of unknown electric potentials in the torso and $A'$ and $b$ represent the adjusted stiffness matrix and the right hand side respectively.

The SCIRun net loads the torso mesh into memory at initialization — each mesh element contains a conductivity tensor that defines how electricity travels through its region of the domain — and reads the electrode meshes from
transformation subnets, which contain UIs to move the electrodes on the torso surface and set the magnitude of the potential difference between the electrodes. Every time the user alters the position or strength of the electrodes, the new stiffness matrix $A'$ and RHS $b$ are re-calculated, and the new system is solved. The gradient, $\nabla \Phi'$, is then taken and passed on to the visualization subnet for rendering (Figure 2).

The study’s design space consists of parameters defining the electrode position and strength, and the performance metric involves 3 dimensions: uniformity (variations in the electric potential in the heart volume), effectiveness (percentage of heart volume whose potential gradient is above the activation threshold), and damage (percentage of heart volume whose potential gradient is above the damage threshold).

The study’s overall objective is to find the points in the design space where the performance metrics yield the best uniformity and effectiveness while minimizing damage to the heart. A user would typically conduct the study in an interactive fashion, using partial results to guide future exploration. We use the following interaction scenarios as a motivation for our experiments:

**Scenario 1: Discovering safe placement of electrodes.** The user initializes the study, requesting a low-resolution version of the Pareto frontier of design space points where the performance metrics satisfy a specified threshold. Once the shape of the frontier is approximately known, the user restricts the explored region to a part of the design space away from the frontier, to introduce an additional degree of safety, and requests a denser sampling of this area to explore current/potential distributions for different electrode placements (Figure 5).

**Scenario 2: Exploring defibrillator efficiency for different body types.** A thicker layer of fat between the electrodes and the heart requires a higher activation voltage for defibrillation to be effective, but the heart can also tolerate a higher voltage before the tissue is damaged. While the most reliable approach would be to model different body types and fat/muscle/bone distributions explicitly, due to a broad range of natural variability, this is difficult. A simpler approach is to change activation and damage thresholds directly, and explore how these changes affect Pareto-optimal electrode placement (Figure 3). For patients with fat tissues covering only part of
the heart, regions of the heart may still be susceptible to normal levels of damage and activation, but the rest of 
the heart is shielded, so the user must be able to restrict the region of the heart where the performance metrics 
are measured.

**Scenario 3: Modifying admissible regions.** Typically, a study is initialized by specifying a relatively large admissible range for observable quantities. As the study progresses the user may choose to adjust admissible regions, with exploration continuing, for example, with lower maximum damage percentage and higher the minimum percentage activation acceptable (Figure 4).

![Figure 3: Pareto points of low (left) and high (right) activation thresholds. Each bar represents one Pareto point. The placement of the bar corresponds to the placement of the Pareto point. The height and color of the bar reflect performance metrics that the user has selected. The frame on the body indicates the region that is being explored. Notice the Pareto points need to move toward the center of the torso as the activation threshold is raised.](image1)

![Figure 4: Scenario 3: Pareto points of large (left), medium (middle), and small (right) admissible damage region. Notice that, in smaller admissible regions, there are fewer Pareto points.](image2)

![Figure 5: Discovering region of interest. After an initial, coarse, sampling of the design space (left), the user restricts the exploration to an area away from the coarse Pareto frontier, and discover a finer set of Pareto points with the restricted region.](image3)
3 SCIRun-based Computational Studies

Our overall approach to implementing SimX functionality on the SCIRun platform is to design a set of components (modules) that can be added to existing SCIRun-based applications to enable user-steered computational studies. While some changes are necessary to convert from an isolated single-process application to a multi-experiment study (most importantly, separation of visualization aspects from the computation and the addition of study-level visualization and steering components), our aim is to provide a set of tools that minimize the required work. For example, most changes to the defibrillator simulation application were related to the dataflow redesign, with the important exception of programming application specific study-level UI/visualization modules. This task, however, was considerably simplified by the facilities provided in the SCIRun toolkit.

3.1 SimX Modules and System Architecture

Integration of SimX within SCIRun requires both additional modules reflecting the various aspects of SimX functionality described in Section 2.3, and a system architecture more suitable for a computational study environment.

Instead of the traditional single multithreaded SCIRun process associated with a steerable application, SimX /SCIRun-based computational studies rely upon a front-end SCIRun process and several instances of back-end SCIRun processes. The front-end process runs on the user’s terminal and permits interaction with the ongoing study like a normal SCIRun session. The back-end processes, optionally run on remote computers, correspond to the simulation container processes, and perform the actual computation of the study.

The front-end process includes a number of modules, including SimXManager and a collection of User Interface (UI) and Visualization related modules, based on UI components and widgets provided by SCIRun. The first module combines the functions performed by the SimX manager and active sampler components, and is responsible for selecting, based on user input, a subset of experiments from the design space to explore, issuing these experiments to simulation container SCIRun processes, collecting the results, and sending the results to the visualization/UI modules. The latter modules are described in more detail in the next section.

The back-end processes augment the existing SCIRun application net with an additional type of SimX module. Arbiter modules are responsible for communicating with the SimXManager modules on the manager process, retrieving experiment parameters, and executing the net that performs the simulation. Arbiter modules require the use of a downstream helper module, which receives the performance metrics of the initiated experiment and passes them back onto the Arbiter module; the latter passes these results to the manager process and receives the next piece of work.

Both the front- and back-end processes also utilize additional Shared Object Layer Reader/Writer modules to read or write data (currently, checkpoints) into the Shared Object Layer. As the SimX implementation matures,
additional modules to handle transformations to and from checkpoints will also be developed.

### 3.2 Converting DefibSim into a Computational Study

The component- and dataflow nature of SimX /SCI Run permits reuse of existing application nets as the basis for larger computational studies with relatively minor code modifications. As an example, consider the defibrillator device simulation described in Section 2.4, whose original net includes four main subnets that (1) allow users to input experiment parameters (thereby steering the experiment); (2) run the experiment; (3) visualize the experiment results; and (4) calculate the experiment’s performance metrics.

Figure 6 shows the SCIRun nets for the simulation container (left) and manager processes (right) built out of the original nets primarily by manipulating module linkages. The simulation container net retains the original simulation-execution (left) and performance metric-extraction (right) subnets, and replaces the user input modules with SimX Arbitrator modules (top right). The simulation subnet is separated from the SimX-connected subnet and is only executed if the Shared Object Layer Reader module (right) fails to find a checkpoint to reuse. An extra connection is provided between the performance metric-extracting subnet and the Arbitrator helper module (bottom right). Finally, the Shared Object Layer Writer module is added to the net (bottom), so that the result of the simulation can be stored.

On the manager process, a SimXManager module (far right) is connected downstream from a steering subnet (top) that generates a specification of a study slice based on user input. The specification is communicated to the SimXManager via a standard SCIRun Matrix port. Three visualisation subnets (bottom) is also connected to the steering subnet. Based on the current study slice, they query the SimXManager to obtain aggregate information about the study. They can then display the aggregate information to the user. They are further described in Subsection 3.3.

The SimXManager receives the specification of a study through a generic SCIRun Matrix port. The matrix is
expected to conform to a specific format and contains the following information: how many dimensions are in the overall design space, how many dimensions are in the current slice that the user is trying to explore, which dimension in the overall design space are fixed, the values they are fixed to, and the range of the non-fixed design space dimensions. While this scheme limits the choice of linear constraints that can be used to define a slice (the user can only fix the value of several of the overall dimension), it is sufficiently generic for the various scenarios of the defibrillator study.

In this section we discuss the facilities our component-based implementation of SimX provides for interactive computational studies.

Interactive experimental studies require both efficient support for feedback and user input at the level of aggregate studies (as opposed to individual experiments), and responsive system-level components that can continuously manage system resources to optimize study progress in the face of changing user-specified objectives. In the SimX system, this support cuts across three sets of components: the visualization and UI components in the front-end, the active sampler functionality embodied in the SimXManager module, and the internal organization of the machine-wide shared object space layer (SISOL).

### 3.3 Visualization and User Interaction Components

Interactive computational studies necessitate a tight two-way coupling of simulation with visualization, i.e. the results are communicated to the visualization modules, and user input is passed back to simulation. What distinguishes the design of these components in computational study infrastructures is that they need to be capable of displaying data from many simulation experiments included in the study, and modifying parameters of a collection of both ongoing and yet to begin experiments according to user input. Needless to say, requiring the user to visualize and/or steer each experiment independently in a study that can involve thousands of experiments is not an adequate solution.

To motivate the organization of SimX interaction components we briefly consider one of the typical interaction scenarios for the defibrillator study, where the user interactively changes the area of interest in which an optimal solution has to be computed. For example, the user may change the area where one of the electrodes can be placed. This action has several effects. First, the visualization of the design space (area where the electrodes are positioned in our case) is updated. Second, the parameters of the area are passed on to the active sampler module, which sends back all available performance data for the new area for visualization, and starts new experiments to sample the missing parts of the area. Both of these actions are performed in the front-end SCIRun manager process.

Study-level visualization and steering in SimX is realized using seven main modules. Three of these modules (and associated auxiliary modules), comprise the user interface, and the other are used for connecting the UI/visualization components to the rest of the system. The overall architecture is problem-independent; however, a significant portion of the functionality contained within UI/visualization components has to be domain specific for efficient user interac-
tion. SCIRun provides an extensive collection of widgets and data structures greatly simplifying construction of such
domain-specific components. More specifically, the interaction modules include the following:

**Design space UI.** This module is responsible for visualizing the currently active slice of the *design space*, and the
performance measures for experiments corresponding to parameter choices within the slice. Typically, design
space slices are naturally associated with areas in physical space, or other natural configuration spaces for the
system. For example, a three-dimensional slice of the design space for the defibrillator study corresponds to
fixing one electrode position, and varying the position of the other electrode on the torso, with a range of
voltages for each position. Most of the interaction of the user with the study is done using this module. The user
can modify areas of interest in the design space, control which performance measures are visualized and choose
the experiment data to examine in the individual experiment UI (Figures 3 and 4).

**Performance space UI.** This module visualizes parts of the *performance* space. By its nature, this space is more
abstract; if more than three performance measures are used, it is hard to visualize the whole space, so only
three-dimensional projections are visualized. Our study possesses only three performance measures, so the
whole space can be seen. Despite the space being abstract, it is useful for clearly understanding the tradeoffs
between different performance measures for a collection of experiments, as well as to specify the admissible
ranges of performance measures (Figure 7).

**Individual experiment visualization.** This component shows the results of a single experiment, if it is desirable to
view them in detail (for example, visualize all damage areas in the heart for a given electrode placement and
voltage). There may be multiple instances of this component initialized dynamically to compare the results of
separate experiments (Figure 7).

**Visualization synchronizer.** As the computational study progresses all three views may have to be updated with new
data. User actions in one UI component may require updates to other components as well as passing requests
upstream to the adaptive sampler. The synchronizer ensures that the three views display data from the same
collection of experiments.

**Active sampler/performance metrics storage.** In the two main UI/visualization components (the design and perfor-
manence space UIs), the user interacts with performance measure data associated with points in the design space.
We assume that the total amount of information is small, so that it can be stored in a non-distributed way, unlike
the cache of checkpoint data maintained in SISOL. This spatially-indexed storage for performance data is asso-
ciated with the active sampler. The storage is queried by the synchronizer whenever the visualizations need to
be updated, and appropriate data is sent to each component.
**SISOL reader.** This is an auxiliary component needed by the individual experiment UI. It communicates with the SISOL servers to obtain the solution to a specific simulation.

### 3.4 Active Sampler

As discussed in Section 2.3, the active sampler converts user specifications of parameter (design) and observation (performance) space domains, the target set, and priority/time target/precision functions into a collection of sample points in the parameter domain for which simulations need to be run. The base active sampler [26] explores the parameter space at increasing levels of refinement; experiments at a higher level are issued only around design space points that dominate their neighboring points (in the sense of achieving a local maxima on at least one performance measure dimension).

We introduce several improvements to the active sampler to support interactive studies. One important new aspect of the active sampler is the notion of a slice of the design space. As the design space is usually relatively high-dimensional (for the defibrillator study, it is five-dimensional) the most natural approach to exploration of this space is to consider two or three-dimensional subspaces of the design space. A *slice* is a part of such subspace, obtained by fixing some of the design parameters or more generally introducing a linear constraint on some of the parameters and setting admissible ranges for other parameters.

For example, in the defibrillator study, a slice is obtained by fixing the position of one of the electrodes, or fixing the direction from one electrode to the other. The position of the other electrode may be restricted to a specific area of the torso. This creates a three-dimensional slice in the parameter space. Exploring a slice is identical to exploring a smaller design space, so the same techniques apply. When the user requests exploration of a new slice, a new active sampler is created. The old sampler is then inactivated, and stored in a data structure in SimXManager. If the user decides to revisit the old slice later on, the sampler will be re-activated again.

In addition to interactively manipulating slice definition, the active sampler is extended to support admissible regions (i.e. constraints on observable values) and modify parameters of the observables.

An example of the first type of functionality is changing the upper bound on the percentage of the heart tissue above the damage threshold (scenario 3); areas where the damage is too high are not refined by the adaptive sampler. Lowering or increasing the activation or damage thresholds, as in interaction scenario 2, is an example of the second type of functionality.

**Study- and experiment-level reuse.** Support for experiment result reuse is one of the most important aspects of the system. Our system supports two types of reuse: *experiment-level reuse* and *study-level reuse*.

Experiment-level reuse is based on the checkpoints stored in the SISOL layer (Section 3.1), which correspond to either the intermediate or final results from previous experiments. Two types of experiment-level reuse are possible.
First, in many situations the result of the simulations at nearby parametric points can be used to initialize the solution process at other points and speed up the simulation (for example, by decreasing the number of iterations needed by a linear or non-linear solver to converge). In the second type of experiment-level reuse, the simulation worker detects when an experiment was already performed for exploration of a different slice. If that is the case, instead of re-running the experiment, it simply requests the solution from the SISOL, and performs evaluation of the performance metric on the data from the old experiment. This type of reuse is common when the definition of the slice is changed by modifying the definition of the performance metrics, e.g., changing the damage threshold.

**Study-level reuse** is based on the idea that slices close to each other have similar Pareto boundaries. When the user requests exploration of a new slice, the active sampler attempts to find a close-by, previously-explored slice, and uses the (possibly incomplete) Pareto optimal points discovered in the old slice to initialize the Pareto boundary search in the current slice.

Finally, all experiment results are shared among active samplers, so that experiments that are issued by different samplers, which can happen in regions where two slices overlap, need not be performed again.

### 3.5 Shared Object Space Layer

A central aspect of the SimX architecture is the machine-wide SISOL shared object space layer, which stores checkpoints and other globally accessed data from previously run experiments. These data are naturally indexed using the parameter space point the experiment corresponds to, so the base SISOL design [26] is structured as a spatially indexed data store. Given requirements of scale and for support for thousands or tens of thousands of experiments, the SISOL implementation is distributed across a multitude of servers, each hosting a partition of the parameter space.

Interactivity, as supported by the user-driven slice-level exploration, imposes two additional challenges, which necessitate improvements to the SISOL design. First, the changing slices of interest imply that the overall design space is explored in a very sparse and unbalanced fashion: this usage pattern is poorly supported by the static partitioning of the base design. Second, different slices implicitly end up associating different importance to different data objects stored in the SISOL layer (e.g., checkpoint data for parameter space points at a greater distance away from the current slice are unlikely to be retrieved): the base SISOL layer ends up using up a large fraction of its memory resources on storing objects that are rarely accessed.

The new SISOL design addresses these challenges. To cope with the possibility of a sparse and unbalanced exploration of the parameter space, SISOL introduces the notion of one or more “directory” servers, which assign a parameter space point to a particular “storage” server; the assignment function ensures that the storage servers see balanced load. The overhead of the directory server lookup is usually small compared to the costs of storing/retrieving the checkpoint data corresponding to an experiment. Even in situations where this might not be the case, the directory
information is amenable to caching on the simulation containers.

The second challenge is dealt with by introducing the explicit notion of a priority value associated with the objects stored in SISOL. Object priorities are recomputed whenever exploration of a new slice is requested (based on the distance of the parameter space point from the region defined by the current slice). These priorities can be used within SISOL in a number of ways, including as in the current implementation, for deciding which objects to replace when a storage server approaches its capacity limit.

![Figure 7: Performance Space UI and Individual Experiment Visualisation. The Performance Space UI (left) shows the Pareto Surface of a design space slice in the 3-dimensional performance space. The Individual Experiment Visualisation (right) shows the region of the heart (highlighted red) that is damaged by a particular placement of electrode.](image)

4 Experimental Evaluation of System Behavior

4.1 Experimental Setup

To quantify the effects of our design decisions on system performance we characterize the behavior of the SimX system using two of the user interaction scenarios for the defibrillator design study described in Section 2.4. In both scenarios, the study runs under active sampler control for some time, until the user changes study goals or constraints. The first scenario – region of interest – models the user changing the admissible region to a smaller, overlapping portion of the design space (representing the user “drilling in” into the interesting portions of the design space). The second scenario – activation threshold – models the user changing the threshold on the activation voltage (representing exploration of defibrillator efficiency for different body types).

To evaluate the responsiveness of the system to such interactive changes, we measure the time it takes for the system to produce new Pareto-optimal or otherwise desirable samples sufficient for the user to proceed. This time depends on many parameters, including on the size of the current experiment database, the degree of change in the parameters of the observables or the target exploration region, and the measure used to decide what sample density is sufficient after user intervention. The experiments below explore the dependence of the overall system performance...
on these parameters, as well as characterize the effects of individual performance optimization strategies.

For all of the experiments, the study was run on a homogenous IBM eServer cluster comprising 256 nodes, each with two 64-bit 2.2 GHz PowerPC 970 processors and 2 GB RAM, interconnected via a Myrinet network (a subset of 32 nodes was used for our experiments). The study involved a single manager process, one SISOL directory server and four SISOL storage server processes serving the checkpoint object set, and varying numbers of simulation container processes, each hosted on a separate physical processor.

4.2 Results

**Overall system performance.** In this set of experiments, we measure the response of the system as a function of two parameters: the amount of work completed before the user changes the study objective, and the degree of change in the performance metrics or design-space region to explore under the new objective. Intuitively, one expects the SimX system response to be better the more experiment history there is to work with, and the smaller the variation in study objective from what the user starts off with.

These parameters are quantified for the two interaction scenarios as follows. The amount of work is measured as the number of experiments issued by the active sampler towards the determination of the Pareto frontier for the initially-specified study objective (upper bounded by the number required for a fully-resolved frontier). The second parameter, the degree of change in the performance metric or area of interest is measured using appropriately normalized quantities. For example, a change in the activation threshold is measured as a percentage of the initial threshold value, and the change in the region of interest is measured as the fraction of the new area that does not overlap the old area.

Tables 1 and 2 show response time measurements for the two interaction scenarios as a function of experiment history and variation in study objectives, respectively. These results are based on the study running on 32 simulation container processes.

Table 1 highlights the advantages of a richer experiment history and the ability of the SimX system to leverage this history for speeding up future study explorations: study runtimes decrease by more than a factor of 3x in both scenarios, as the amount of history increases.

Table 2 similarly shows the ability of the SimX system to reuse previous results in computing the Pareto frontier for a new objective that is “close” to the current one. As expected, the largest runtime improvements, up to 4x, are seen in situations (in both interaction scenarios) where the degree of change in the performance metrics or design-space region is small. What is interesting is that even for relatively large changes, SimX is able to achieve substantial reuse.

Note that the magnitude of the improvements reflected here are limited by the length of the studies we used in our experiments; typical longer-running studies should realize additional benefits.
Table 1: Response time for study interaction scenarios, as a function of experiment history.

<table>
<thead>
<tr>
<th>SCENARIO</th>
<th>RESPONSE TIME (S)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Experiments issued before objective change</td>
</tr>
<tr>
<td>Region of interest</td>
<td>Base</td>
</tr>
<tr>
<td></td>
<td>100   200  300  400  500  600</td>
</tr>
<tr>
<td></td>
<td>151.829 126.55 113.51 92.40 80.35 63.81 47.71</td>
</tr>
<tr>
<td>Activation threshold</td>
<td>Base</td>
</tr>
<tr>
<td></td>
<td>100   200  300  400  500  519</td>
</tr>
<tr>
<td></td>
<td>124.121 83.6232 72.8647 45.7605 45.7122 28.179 29.1362</td>
</tr>
</tbody>
</table>

Table 2: Response time for study interaction scenarios, as a function of variation in study objectives.

<table>
<thead>
<tr>
<th>SCENARIO</th>
<th>RESPONSE TIME (S)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>% overlap in region of interest</td>
</tr>
<tr>
<td>Region of interest</td>
<td>Base</td>
</tr>
<tr>
<td></td>
<td>0%  6.25%  25%  43.7%  62.5%  81.3%</td>
</tr>
<tr>
<td></td>
<td>151.829 152.23 158.897 125.277 102.298 89.5233 47.3706</td>
</tr>
<tr>
<td>Activation threshold</td>
<td>Base</td>
</tr>
<tr>
<td></td>
<td>% change in activation threshold</td>
</tr>
<tr>
<td></td>
<td>45%  36%  27%  18%  9%</td>
</tr>
<tr>
<td></td>
<td>124.121 75.3981 76.0747 54.9542 45.555 29.1362</td>
</tr>
</tbody>
</table>

B: Active sampler performance. To measure the effects of the active sampler on system performance, we run experiments similar to the above, and measure the difference in performance with respect to the simple grid sampler. An important feature enabled by the adaptive sampler is study level reuse, i.e. the ability to use the results of the completed experiments even if the objective functions have changed. The improvement due to study-level reuse are demonstrated using the second scenario, varying the amount of work done before the change of objective.

Table 3: Comparing sweep sampler and active sampler

<table>
<thead>
<tr>
<th>SAMPLER TYPE</th>
<th>RESPONSE TIME (S)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Experiments issued before objective change</td>
</tr>
<tr>
<td>Sweep Sampler</td>
<td>Base</td>
</tr>
<tr>
<td></td>
<td>100   200  300  400  500  600</td>
</tr>
<tr>
<td></td>
<td>793.00 649.849 635.527 618.632 601.984 592.989 570.543</td>
</tr>
<tr>
<td>Active Sampler</td>
<td>124.121 83.6232 72.8647 45.7605 45.7122 28.179 29.1362</td>
</tr>
</tbody>
</table>

Table 3 shows that, although sweep sampler is able to take advantage of experiment-level reuse, it is still 6 to 19 times slower than the Active Sampler.

C: Scalability. We measure scalability of the system by fixing a user action in one of the usage scenarios, and vary the number of simulation workers dedicated to the study, and observe how the system’s response time changes. We chose the second scenario (changing the activation threshold by 9%), and fully completing the study before the objective change. The study is done on 1, 2, 4, 8, 16, and 32 simulation workers.

As shown in Table 4, the system scales reasonably well up to 32 processors. The 10-20% departure from perfect
Table 4: Scalability as measured by response time

<table>
<thead>
<tr>
<th>Num of Simulation workers</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reponse time (in secs)</td>
<td>393.138</td>
<td>213.951</td>
<td>120.585</td>
<td>81.8655</td>
<td>53.5305</td>
<td>29.1362</td>
</tr>
</tbody>
</table>

scaling can be explained by the algorithmic overhead already described in [26].

5 Next Steps

We have demonstrated that, by leveraging SimX’s permeable system-application interface and SCIRun’s component-based dataflow framework, it is possible to create a scalable, modular, and interactive problem solving environment where the user can steer and visualize entire computational studies that is being computed on multiple compute nodes of a cluster.

We have identified several avenues for further development of the SimX/SCIRun system that we summarize below.

Advanced checkpoint reuse. We are working on a general-purpose checkpointing functionality for SCIRun, which would allow to save/restore arbitrary dataflow link state. This allows easy saving of incomplete simulation states for reuse.

Active sampling schemes The 3D version of the study shows that active sampling is less effective in higher dimensions. Our analysis shows that this happens because the sampler resolves all parameter dimensions equally, even when the performance values exhibit different sensitivity. This suggests the importance of introducing user-defined application or domain specific sampling strategies.

Caching in Shared Object Layer Our experiments show that with larger numbers of simulation processes, the overhead of checkpoint transfer to/from the layer becomes a significant contributor to the overall runtime. If a caching implementation of the shared object layer is used, where multiple copies of an object can be cached, the simulation processes can keep a cache of recently used objects (in addition or in lieu of writing it to the server or the home node) and potentially save network traffic.

Other important development directions include allowing individual experiments to run on multiple processors and developing efficient scheduling strategies for this purpose, and developing user-level tools for computational study initial setup.

References


