

Case Study: Visual Debugging Of Cluster Hardware

Patricia Crossno¹

Rena Haynes²

Sandia National Laboratories³

Abstract

This paper presents a novel use of visualization applied to debugging the Cplant™ cluster hardware at Sandia National Laboratories. As commodity cluster systems grow in popularity and grow in size, tracking component failures within the hardware will become more and more difficult. We have developed a tool that facilitates visual debugging of errors within the switches and cables connecting the processors. Combining an abstract system model with color-coding for both error and job information enables failing components to be identified.

CR Categories and Subject Descriptors: I.3.8 [Computer Graphics]: Applications; J.2 [Physical Sciences and Engineering] Electronics; C.4 [Performance of Systems] Modeling techniques – Performance attributes.

Additional Keywords: visual debugging, hardware modeling, design analysis, and performance optimization.

1 INTRODUCTION

The motivation for this work stemmed from our inability to diagnose hardware errors based on lists of error counts, such as for bad packets or bad routes, for each port of each switch in our system over a period of time. Trying to map error information to system schematics by hand was tedious and slow, plus it lacked the ability to dynamically portray error propagation over time. Without being able to correlate errors with our system topology, it was unclear whether the errors occurred at random or if they were concentrated around particular switches or network cables.

Applying some of the ideas from our previous work in debugging particle systems [3], we developed a tool for visually debugging our cluster hardware. The central concept is that the human visual system's pattern recognition skills can be employed to see relationships between error counts and hardware components when they are combined in a visual abstraction of the system.

The tool allows us to visualize a model of the switch hardware and network connections combined with error counts for each port, the jobs running on each processor, and the routes used in the communications between processors. With this tool we have been able to see patterns of error propagation along routes originating at processor ports. We have also been able to go back and detect a faulty cable that was initially diagnosed through other means.

2 RELATED WORK

Although we have not found any previous work that uses visualization to debug cluster hardware, using visualization to

debug or do performance analysis of parallel programs is relevant. The *Rivet* system performs analysis and visualization of parallel applications in a shared memory environment [1]. Browne et al created a directed graph representation for parallel programs to simplify parallel programming [2]. *The Interactive Visualization Debugger* integrates debugging, performance analysis, and data visualization for message passing parallel applications [4]. *Devise* is a generic integrated performance analysis and visualization system that has been coupled with the *Paradyn Parallel Performance Tool* [5]. The *MAD* environment is a toolset for parallel program debugging [6]. Zhang, Hintz, and Ma use graph formalisms and notation to visualize parallel programs and their execution [8].

3 IMPLEMENTATION

The tool is implemented in C using OpenGL as the graphics API. The user interface is implemented using GLUT version 2.0, a GLUT-Based User Interface Library written by Paul Rademacher that is freely available from UNC [7]. The tool is platform independent and runs on IRIX, LINUX and NT systems. A snapshot of the tool showing the user interface is shown in Figure 6 on the color plate.

3.1 Model

Schematics describing the switch topology form the basis of the mental model used by people debugging the cluster hardware. So the core of the system is a representation of this model. However, the model is not static. We already have a number of different Cplant™ system configurations of various sizes within Sandia, plus, we want the tool to migrate to newer systems as they come online.

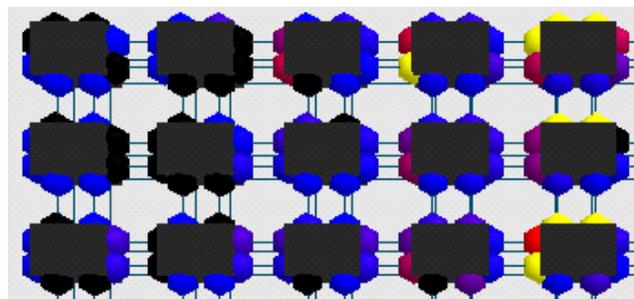


Figure 1: This is the upper left section of a planar switch layout where each switch (black box) has eight network ports (spheres on the box faces), each of which is colored by a total error count. Network links are drawn between ports. Since the topology of the network is really a torus, long links are drawn behind the plane of switches connecting the first and last switches in each row and column. At this angle from the front, the wrap-around links appear to be a second link between each pair of ports.

So model information is input to the tool at runtime. Common to all models is the idea of a switch. Each switch has

¹pjcross@sandia.gov, ²rahayne@sandia.gov

³P.O. Box 5800, Albuquerque NM 87185-0822

sixteen ports, with some of the ports connected to processors and some of the ports connected to the network. To reduce visual clutter, all of the ports are not displayed simultaneously. Radio buttons select a port display mode, switching between processor and network ports. The network connections between ports serve as a visual cue as to which type of port is currently being displayed. Although only the ports are color coded, it is important to show the switches to provide context.

Switches are laid out in a planar fashion, as is shown in Figure 1. Network connections wrap at the tops and bottoms of both rows and columns. Given that the connections actually form a torus, we provide an alternate display mode that reflects this topology, as is shown in Figure 2. However, the users prefer the planar layout with wrap-around links drawn behind the switch plane. Probably this is because it more closely matches their mental model from the schematics.

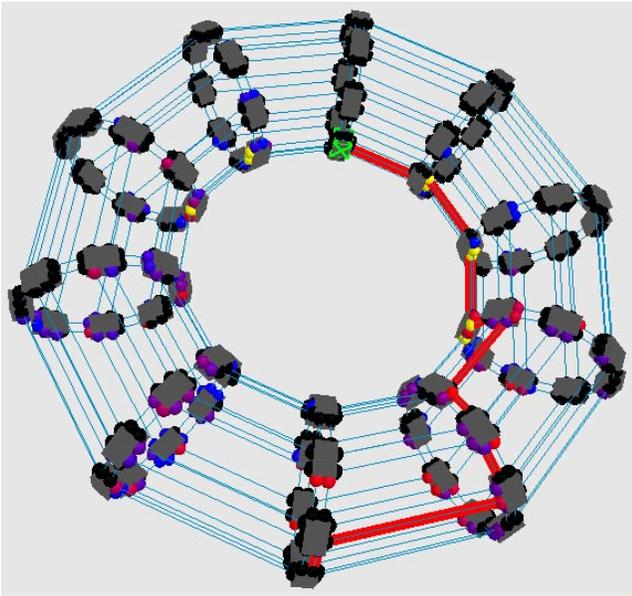


Figure 2: Torus layout showing true network connection topology. The thick, red line is a route between 2 ports.

In larger systems, multiple planes are stacked, as is shown in Figure 3. Although on smaller systems the ports are evenly divided between processor and network ports, in multi-plane systems more ports must service the network connections to enable communication between planes. This means that there are twelve network ports (two per switch face) and only four processor ports. Additionally, large models have an additional hierarchical grouping of pairs of switch rows into a meta-switch called an XBAR. The display provides a reference to this structure by drawing a yellow wire-frame box around each XBAR. Note there is no communication wrap-around in the Z direction.

Given the density of the switches in Figure 3, it is difficult to see the ports on interior switches. To reduce the visual complexity, we have removed all switches whose ports do not have any errors. The top two rows of switches in the last plane, along with two individual switches in the first plane, and the second and third rows of the second plane have all been removed. But this is still too cluttered to see significant detail, so we provide checkboxes to turn individual planes on and off. Additionally, providing interactive rotation, translation, and zoom enables the user to manipulate the model to focus on areas of interest.

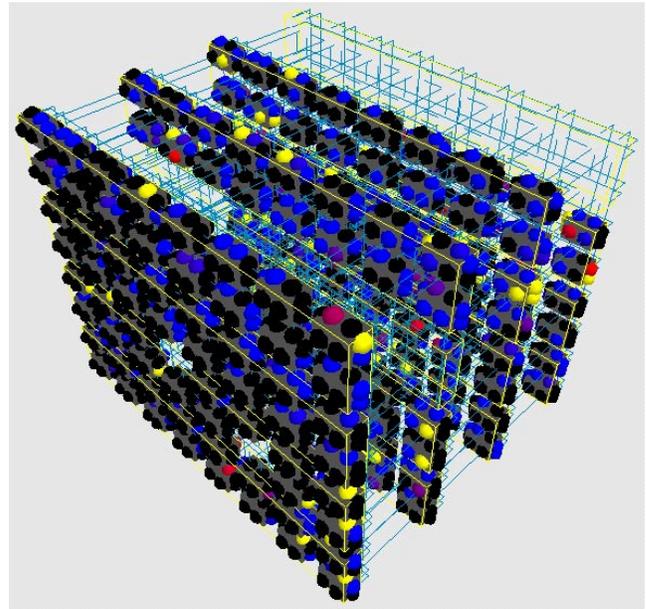


Figure 3: Large system model showing multiple planes of switches with twelve network ports per switch. Pairs of switch rows are grouped into XBAR meta-switches.

The numbers of process versus network ports and the connectivity and labeling of the ports and the switches are all contained in the model file. The colors of the switches and the network connections may be specified through an initialization file, or default colors (dark gray and black, respectively) will be used. The labels for both switches and ports are then used to identify model components. When objects in the model are picked, the objects are highlighted with a bright green wire frame overlay and the corresponding labels are displayed. Inversely, labels can be typed in to locate components, which are then highlighted.

When reading job or error information, labels in the input files are parsed and used to find the corresponding data structures for those ports. By using the labels as the identification mechanism linking the model and the input files, we avoid any implicit assumptions about model component naming conventions and we allow the users to operate using their own name space.

3.2 Errors

Once the model has been read into the tool, error information can be added to the data structures for each port. Error data is input through a file consisting of a list of port labels, each of which is followed by a series of error counts for each time interval. The initial value in the file provides the number of time samples for each port.

The error files are generated by software that periodically polls counters for each port on each switch. The counters contain the number of occurrences for events such as bad packets, good packets, timeouts, bad routes, dead routes and uptime. The difference between the current count and the previous count is then recorded as the error value for this time step. Typically, the switches are polled at frequencies ranging between once an hour and twice a day.

The error counts are displayed in the model by coloring the ports. The error counts can either be viewed either statically or dynamically through animation. Statically, the error displayed for each port is the accumulated total over all of the time steps. In

animation mode, the port color represents either a running total of the error counts over all of the previous time steps, or just the error count for the current time step.

Since the range of values that error counts can assume depends upon the type of errors being viewed and the frequency and duration of the polling, the color mappings depend upon a user-defined range of values. The user controls the color-to-value mapping through selection of a minimum value and a maximum value. Values below the minimum are not of interest, so they are mapped to black to blend into the switch. Values above the maximum are of high interest, so they are mapped to yellow to stand out. If black and yellow do not suit the user, different colors for displaying values below or above the range can be specified in the initialization file. Values within the range are mapped to the range of colors between blue, the minimum value, and red, the maximum value. Since colors do not have implicit meanings, the blue to red continuum was chosen because blue can be thought of as a cool, or low, value, and red can be thought of as a hot, or high, value.

3.3 Jobs

In order to understand which processors are likely to be communicating with one another, it is necessary to have a snapshot of the job identifiers for each of the processor ports coinciding with the times that error counts were polled. The jobs file has a similar format to that of the error file. The file consists of a list of processor port labels, each followed by a list of job identifiers for the job running at each polling time. If no job is running, an identifier of zero is used as a flag. Idle ports are colored black when the port display mode is set to color ports by job identifier. All other job identifiers are assigned a random color.

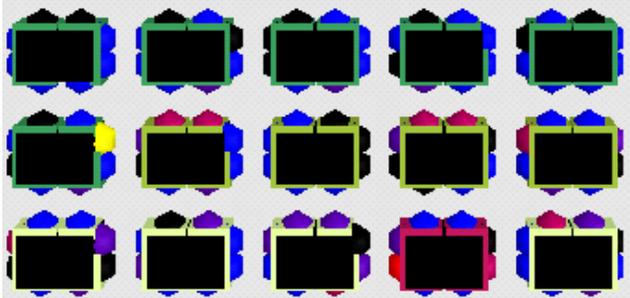


Figure 4: Total cumulative error is displayed in the color of the process ports. Job id is shown in the color of the shallow box around the base of each port. Idle processors do not have a colored box.

Initially, we tried to use shape to encode the job information so as to avoid overloading color. However, the number of different distinguishable shapes was insufficient to encode more than a handful of different jobs. And even using a small set of unique shapes, the job processor groupings could not be seen at a glance, but required some concentration. So we tried coloring the ports by job and toggling back and forth between displaying the errors and displaying the jobs. The groups of processors running the same job could be easily identified, but the toggling was distracting and confusing.

The job identifiers needed to be displayed concurrently with the error information. Rather than replacing the two earlier display modes, *errors* or *jobs*, we added a third option of *both*. In this mode we draw a colored, shallow box around the base of each processor port's sphere to display the job color. The three-

dimensionality of the box allows the job color to be visible from a variety of viewpoints, yet it does not block the visibility of the port color. Idle ports do not have any box.

3.4 Routing

As we began to suspect that errors were passed from switch to switch along a communication route, we needed to include route information into our visualization. We expanded our picking capability to select a start and an end for the route. Either ports or switches can be selected as either sources or destinations. A radio button designates the picking function: *select*, *start route*, or *end route*. Just as with normal picking, a bright green wire frame overlay is used to highlight the selection. Typed in labels can replace picking to select endpoints.

A *draw route* checkbox enables the route to be drawn as a thick red line along the corresponding network links. A different route color can be specified through the initialization file. Figure 6 presents a view from the backside of the switches to show a route that uses a wraparound link between a single port and all the ports on a switch (on the top row with the green X on its face). The same route is shown in Figure 2 using the *torus* layout. This layout presents a clearer picture since the switches in the wrap around link are actually adjacent in the network topology.

Routing information is contained in a set of files. Given the starting port, a corresponding file is opened and a line corresponding to the destination is read. This line encodes a series of hops, in terms of ports, between the source and destination. If a switch is selected for either the source or the destination, routes are drawn between each of the switch's ports and the other end of the route. Naming conventions and directory information for the route files are contained in the model file.

3.5 Animation

The animation functions are *rewind*, *back step*, *forward step*, and *play*, which are accessed as buttons in the GUI and are modeled on a VCR interface. Random access to individual time steps is available through a type-in window, which also displays the current time step when the button functions are used. When *play* is selected, the animation starts from the current time step and stops with the last time step. The speed of the animation is controlled by the spinner *update pause*, which is in seconds.

The error data, the job data, or both can be viewed dynamically. The color-coding of the error data depends on the range selected and whether *running* or *slice* is selected for the error display (see 3.2). If the error display is set to *total* when animation is started, it is automatically reset to *running*.

Although stepping forward and backward through a time sequence has proved to be useful, we have found that the play function is almost never used since the users prefer precise control.

4 RESULTS

Our first test of the tool was to go back and see if we could detect a known hardware error from data that had been gathered during a failure. We had had a faulty cable that was discovered only after replacing a switch failed to resolve the problem. Looking back at our logs and stepping through the error data, we can see the origins of the error in a single network port, shown in yellow in the first frame of the time series in Figure 7. In successive frames, the error propagates along the routes connecting the two highlighted switches at the bottom (source) and top (destination) of the column of switches. The errors originate in the link prior to

the yellow port, though they are only seen in the error counts on ports downstream from the source. With each successive time step the downstream error counts grow as bad packets continue to flow from the bad cable.

In our next example, we had a situation where the entire cluster was dying and we did not know why. We were able to track down the source of the problem using our tool. It turned out that four switches from the cluster had been designated a 32 node *virtual machine*, though the switches and nodes remained physically connected to the larger cluster. The people working on the virtual machine decided to reboot their part of the system when a node became hung. Rebooting caused high error counts on the processor ports of the switches involved as shown in Figure 5. These errors were then propagated along the route shown in Figure 6 back to the highlighted switch in the top row. Amongst the nodes on this switch is a central node which does the scheduling of jobs. The corrupted messages reaching this node led to the failure of the scheduler for the entire cluster, thereby causing a cluster-wide failure. Correlating the error counts, the route, and the switches associated with the virtual machine enabled us to discover the cause.

Further testing outside of our tool has found that the source of the problem comes from how the nodes are powered down. An unload must be done to quiet the state of line drivers on the switch adapter, thereby minimizing any spurious activity that they would subsequently transmit. However, since the node is hung, this unload cannot be initiated. We expect to file a problem report with the network vendor in the near future.

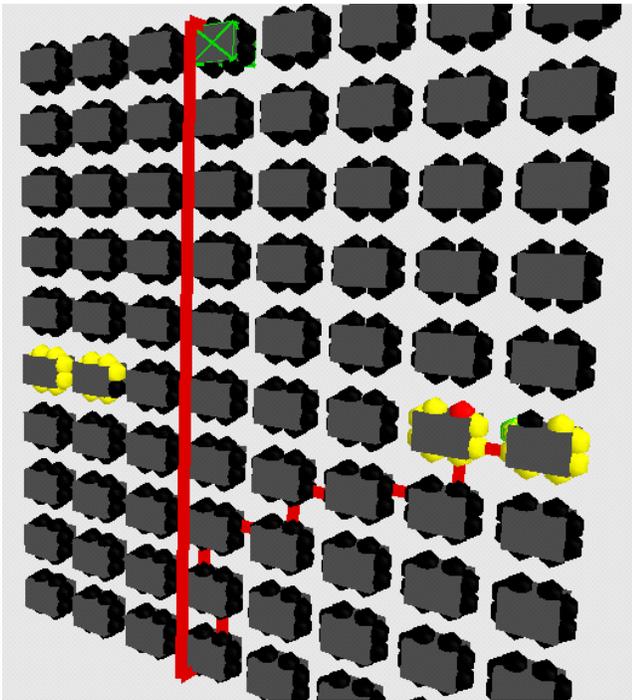


Figure 5: Errors on processor ports in virtual machine.

5 CONCLUSIONS

Although we initially created this tool to help us debug hardware errors on our cluster, we have discovered that this tool provides a powerful way to view all sorts of data and has provided insight into a number of unanticipated areas. Therefore, we have

expanded our view of the tool's utility to include performance optimization and design analysis.

As an example of performance analysis, visualizing the job distribution of the nodes has led us to change how our scheduler works. Now nodes in the same job will be allocated near each other physically, rather than just being logically contiguous. Seeing the spatial correlation between the switches and switch labels has also led to changes in system configuration design.

Another unexpected insight came from some testing we did where we took the entire cluster and simultaneously started a number of jobs. Their exclusive task was to divide the nodes in the job between senders and receivers and to send a large number of messages between them. All the jobs were essentially identical, yet some had high message throughputs, while others did not. Looking at the routing, we could see that certain switches were common to multiple routes and acted as a bottleneck. Now we are examining how we design our routes to eliminate this problem.

6 ACKNOWLEDGMENTS

We would like to thank Eric Russell for providing the error data. The DOE Mathematics, Information, and Computer Science Office funded part of this research. The work was performed at Sandia National Laboratories. Sandia is a multi-program laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under Contract DE-AC04-94AL85000.

References

- [1] Robert Bosch, Chris Stolte, Gordon Stoll, Mendel Rosenblum, and Pat Hanrahan. Performance Analysis and Visualization of Parallel Systems Using SimOS and Rivet: A Case Study. In *Proceedings of the Sixth International Symposium on High-Performance Computer Architecture*, pages 360-371, January 2000.
- [2] James C. Browne, Syed I. Hyder, Jack Dongarra, Keith Moore, and Peter Newton. Visual Programming and Debugging for Parallel Computing. *IEEE Parallel & Distributed Technology: Systems & Applications*, Vol. 3, Issue 1, pages 75-83, 1995.
- [3] Patricia Crossno and Edward Angel. Visual Debugging of Visualization Software: A Case Study for Particle Systems. In *Proceedings of Visualization '99*, pages 417-420, October 1999.
- [4] Ming C. Hao, Alan H. Karp, Milon Mackey, Vineet Singh, and Jane Chien. On-the-Fly Visualization and Debugging of Parallel Programs. In *Proceedings International Workshop on Modeling, Analysis, and Simulation of Computer Telecommunication Systems*, pages 386-391, 1994.
- [5] Karen L. Karavanic, Jussi Myllymaki, Miron Livny, and Barton P. Miller. Integrated Visualization of Parallel Performance Data. *Parallel Computing*, Vol. 23, pages 181-198, 1997.
- [6] D. Kranzlmüller, S. Grabner, and J. Volkert. Debugging with the MAD Environment. *Journal of Parallel Computing*, Vol. 23, No. 1-2, pages 199-217, April 1997.
- [7] Paul Rademacher. GLUI: A GLUT-Based User Interface Library. <http://www.cs.unc.edu/~rademach/glui>, June 1999.
- [8] Kang Zhang, Tom Hintz, and Xianwu Ma. The Role of Graphics in Parallel Program Development. *Journal of Visual Languages and Computing*, Vol. 10, No. 3, pages 215-243, June 1999.

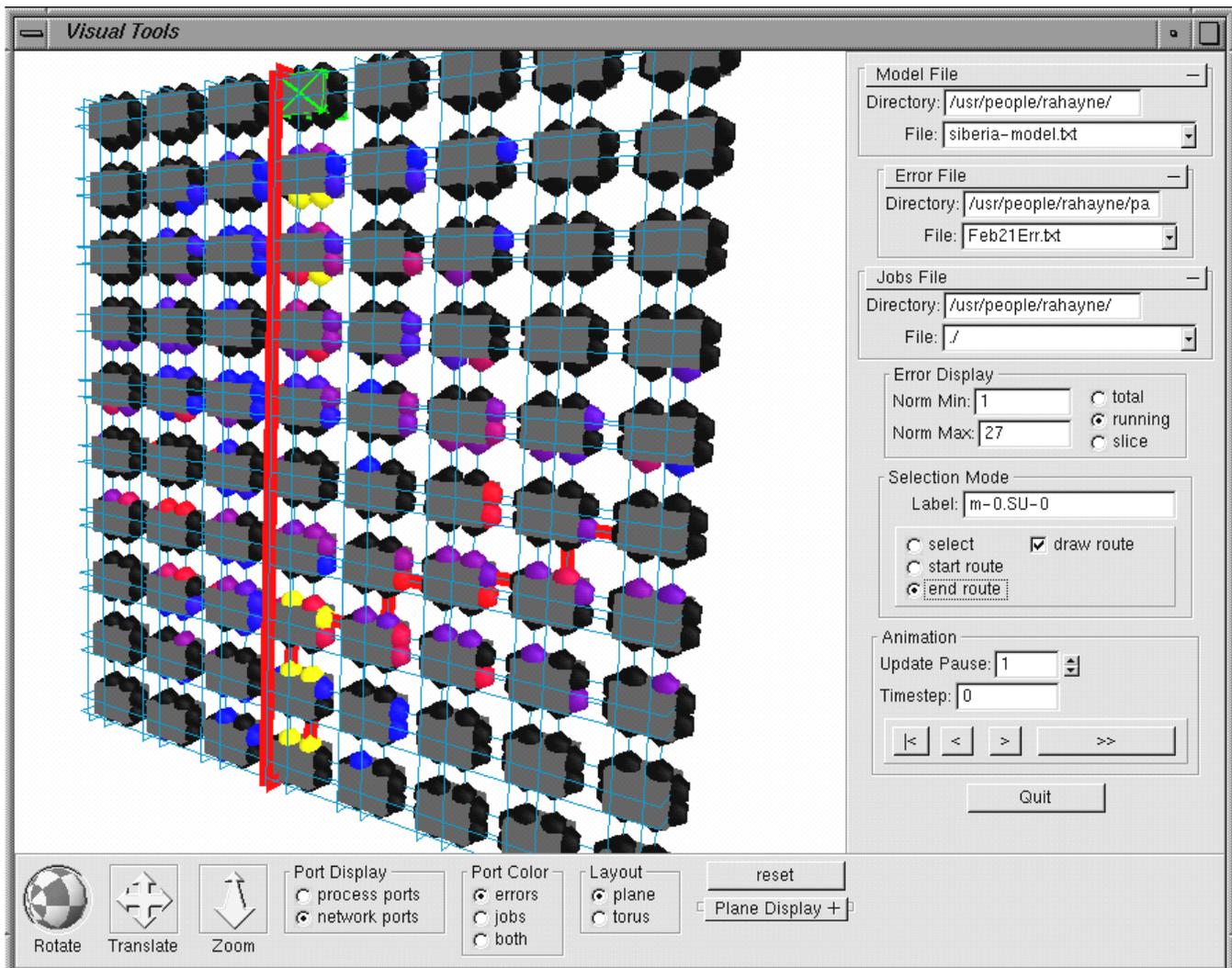


Figure 6: User interface for the tool. Model viewed from behind to show wrap-around links. Display shows the network ports with the same route as in Figure 5 to visualize the error propagation along the route due to the reboot of the *virtual machine*.

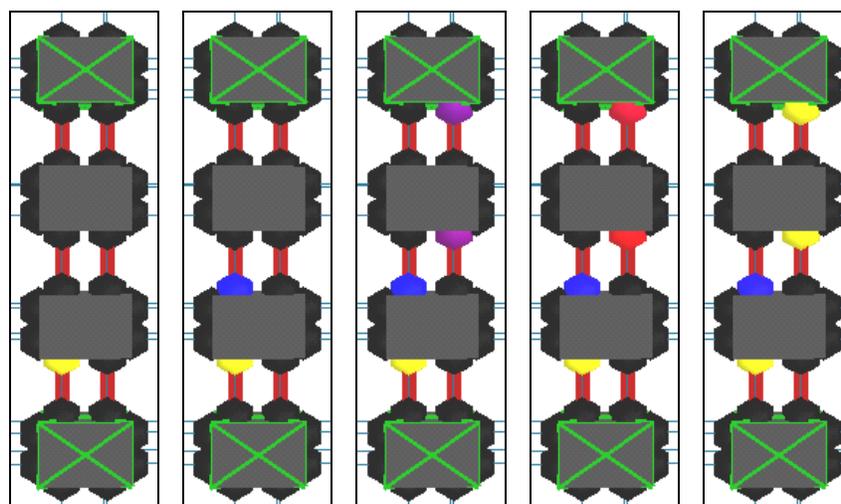


Figure 7: Five time steps showing error propagation due to a faulty cable (link below lower left yellow port) along routes connecting switch at bottom of column to switch at top of column.