# Traceview: A Trace Visualization Tool

ALLEN D. MALONY, DAVID H. HAMMERSLAG, and DAVID J. JABLONOWSKI, *Center for Supercomputing Research and Development*

◆ *In a general-purpose, reusable platform, Traceview implements the trace-management and I/O features usually found in special-purpose trace-analysis systems.*

Although they contain much performance detail, large trace files capturing logical or physical actions taken by a program are difficult to use when analyzing a system's performance. The manual effort required to manipulate trace files, including creating graphical presentations, can be daunting, requiring some automatic support for trace analysis and visualization.

Trace-based performance visualization gives you an intuitive understanding that is often more useful than a textual statistical profile.[1-3] However, systems that support performance visualization often only accept trace input of a specific type, conforming to a particular execution paradigm or generated from a particular system context.[4-6] Furthermore, the displays used can be inadequate to show the time-dependent behavior of arbitrary data values that might be associated with different events in the trace.[6]

A general-purpose trace-visualization system can incorporate common aspects of trace processing and display. However, the gains in reusability come at the expense of specificity in trace-data analysis, because such a system must use a simplified event-interpretation model. Whether this trade-off is a liability will depend on the trace-visualization application.

In this article, we describe the design, development, and application of Traceview, a general-purpose trace-visualization tool. We seek to identify the aspects of trace visualization that can be incorporated into a reusable tool and evaluate the trade-off in general-purpose design versus semantically based, detailed trace-data analysis.

## ARCHITECTURE AND FUNCTION

The architecture for a general-purpose


0740-7459/91/0900/0019/$01.00 © IEEE

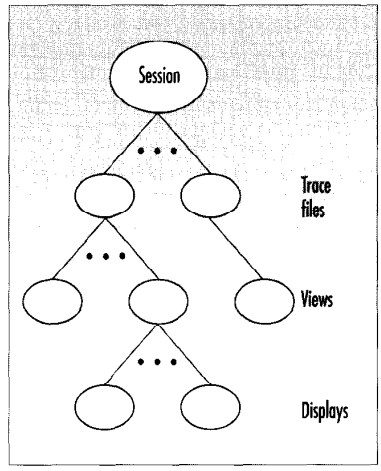


*Figure 1. Traceview session model.*

trace-visualization tool must be flexible enough to let you select analysis and display alternatives. However, it should also provide a structure rigid enough to let you build on the resources of the tool and extend the base analysis and display methods.

Such an architecture cannot support all trace-visualization models, but it should support most — especially those used for the simple visualization problems that occur most frequently. Extension mechanisms should provide an easy customization path for more complex cases.

We based Traceview's general-purpose architecture on the concept of a trace-visualization session. Figure 1 shows the hierarchical tree structure of a Traceview session, which involves trace files, views, and displays.

First you specify a set of trace files to visualize. For each trace file, you can define a set of views. A view defines a trace subregion by setting a beginning and an ending point, and by event filtering. For each view, you can create a set of displays. Although the session paradigm precludes displays that combine data from multiple traces, it supports multiple simultaneous displays. You can use multiple displays to compare data from several trace files.

**Session management.** As Figure 2 shows, Traceview's session manager saves and restores each session's configuration and coordinates the trace, view, and display managers. The session manager lets you save a session configuration to external storage for later retrieval of the same visualization environment. This saves work because Traceview visualization sessions can be quite complex, with many trace files opened and many views defined on each trace.

At any point while you use Traceview, the session manager defines the current session as the set of open trace files, the set of defined views for each trace, and the set of displays for each view. However, the current displays for each view are not part of the session configuration. They are defined only for the current Traceview invocation. Our initial reasoning was that saving the total display state would require too much space. Now we are determining if display saving can be accomplished with only part of the display state.

For each open trace, the session manager saves the name of the file containing the trace data, the number of defined views, and the information necessary to reconstruct each view. The session manager assumes that a session-configuration file is consistent with the data in the designated traces. In future implementations, the session manager will record the modification dates of trace files in the saved configuration and check the dates when it restores the configuration.

The session manager lets you merge multiple session configurations into the current session. If all trace files are distinct, merging is simple — basically, it is additive. When there are conflicting traceview combinations, you are prompted to resolve the conflict by selecting one alternative over another or by renaming entries.

Figure 3 shows how a session appears to a user. Open files are listed in the Files window, defined views for the selected file in the Views window, and created displays for the selected view in the Displays window. When you select a trace file from the open files list, Traceview automatically updates the views list to show the corresponding defined views. Similarly, when you select a view, Traceview updates the displays list to show the view displays you have created. You can add or delete files, views, and displays at any time. The number of trace files, view specifications, and displays is limited only by the memory available to store the pertinent session information.

**Trace files.** Traceview processes event traces. An event is a recorded instance of some logical action. We intentionally give only general descriptions of events because Traceview makes no semantic interpretation of the actions that the events represent. It assumes that each event is time-stamped merely to establish an ordering relation (usually a time ordering) among the events.

The trace file is divided into two parts: an ASCII header and binary trace data, which is a time-sequenced list of trace events. Each event reflects the instance of some action taking place during computation. Traceview interprets this action as a state transition. Each event recorded in the trace includes encodings of the state

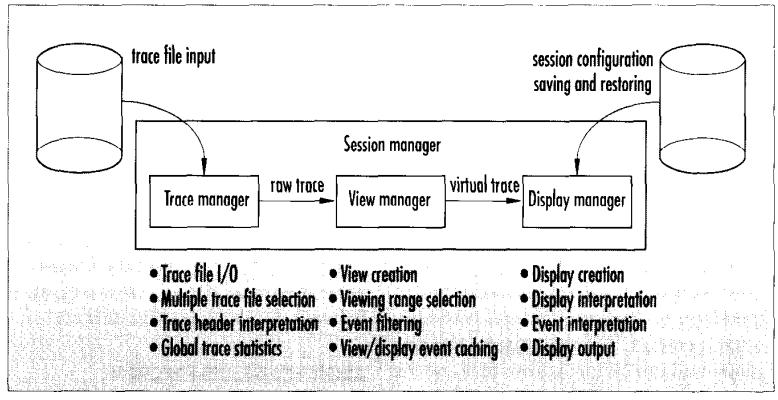


*Figure 2. Traceview architecture.*

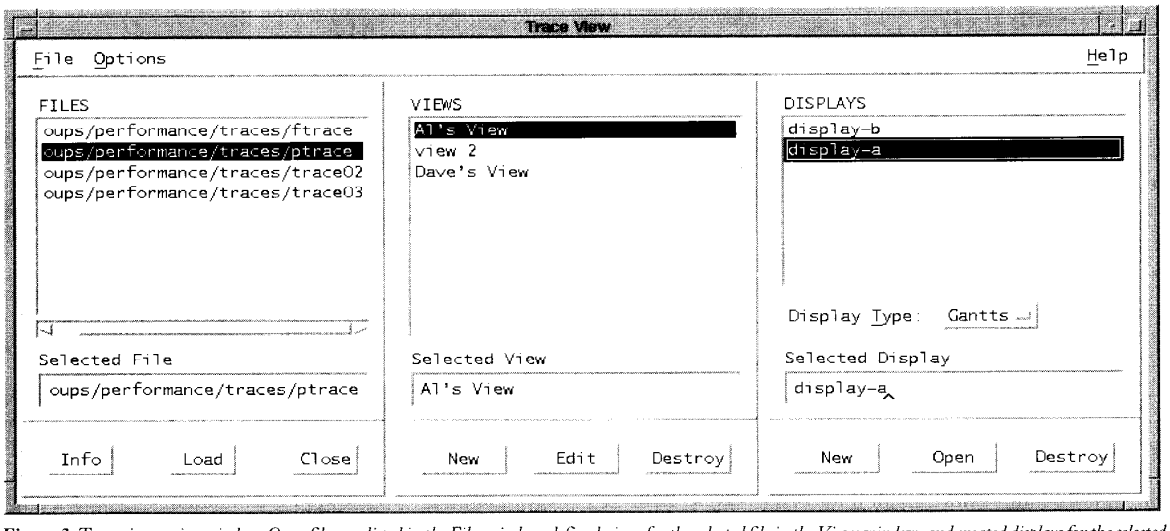


*Figure 3. Traceview session window. Open files are listed in the Files window, defined views for the selected file in the Views window, and created displays for the selected view in the Displays window.*

being exited and the state being entered, an event type, and a time stamp. Events may also include supplemental data fields. Events within a file are homogeneous in their format: Each has the same number of data fields associated with it. (Future implementations will allow variable-size data fields dependent on the state.) Each data field typically represents some numeric event metric.

The trace-file header specifies the number of data fields associated with each event transition and how the data is labeled when presented to the user. It also provides flags to control display customization, a directory of names to use for the states, and an optional index into the event data. Summary information in the header includes the number of events in the trace and the total time represented by the tracc data.

**Trace management.** Working with this trace-file format, the trace manager

- opens trace files,
- interprets the trace-file header,
- calculates global trace statistics,
- reads events from open trace files, and
- closes trace files (freeing storage allocated when the trace was opened).

The trace manager also provides a graphical user interface. You select traces to open through a standard dialogue. The trace manager presents the list of open files in the left window of the main Traceview window, as shown in Figure 3, and presents summary information in the display shown in Figure 4.

**View management.** To define multiple views on each trace, you use the view manager. A view definition consists of

- a starting time in the trace,
- an ending time, and
- a list of names of events to be excluded from the trace.

The view manager applies a view definition to a trace to produce a virtual trace,

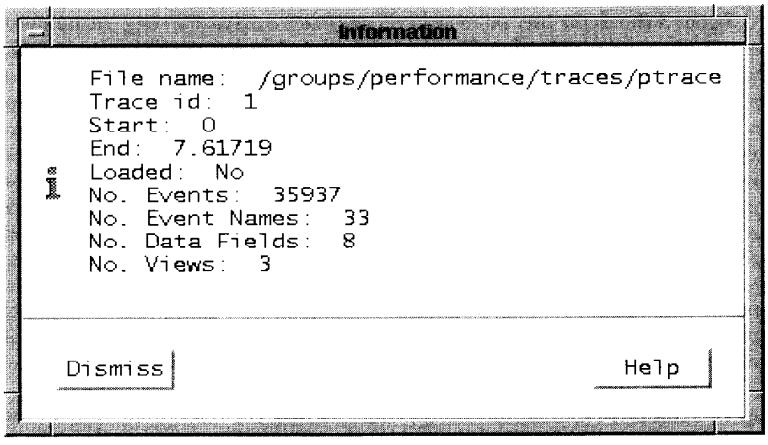


*Figure 4. Trace information window showing summary information for the trace file highlighted in Figure 3.*

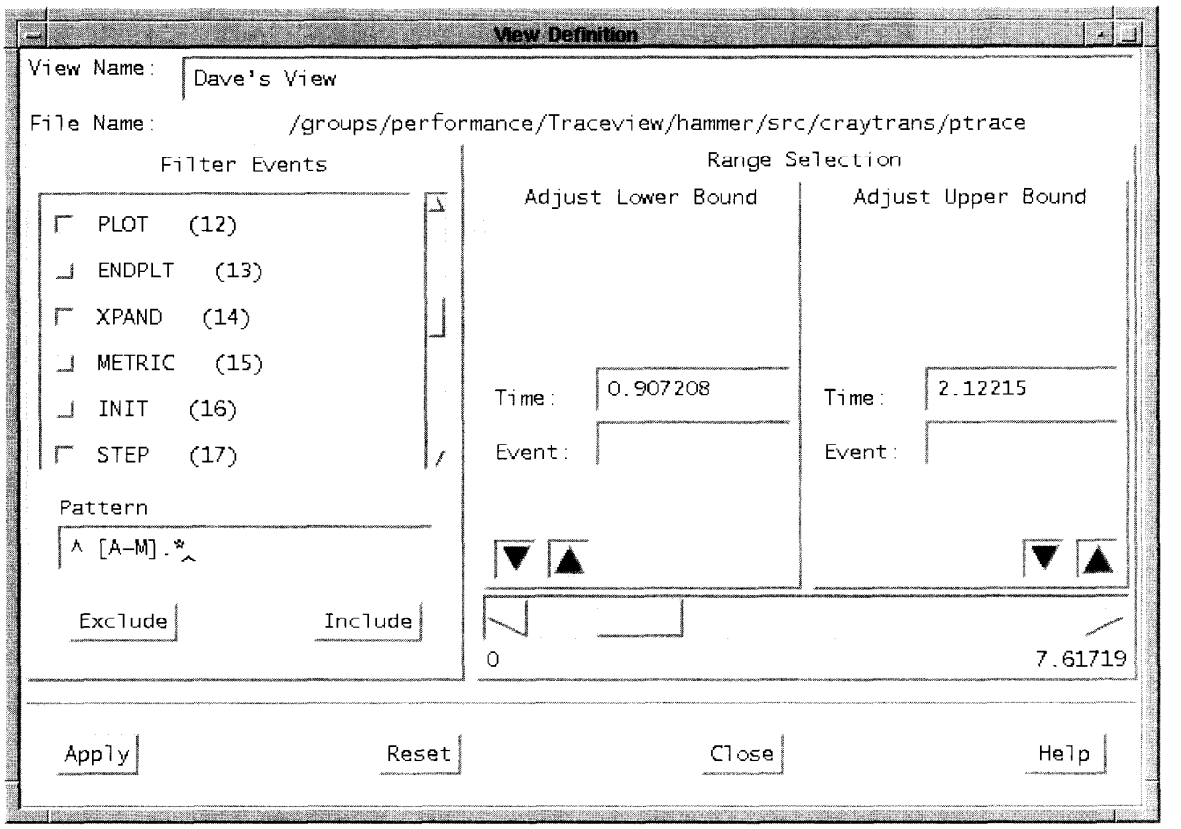


***Figure 5.*** *View-definition dialogue window. In the range selection portion, you select or adjust a lower and upper bound of time or events. To filter events, you toggle event names individually or collectively in the event filter list. And you change the effect of pattern matching using the Exclude and Include buttons.*

which is derived from the actual trace. The view manager first discards any events that occur before the view starting time or after the view ending time. Then it filters the remaining events to remove the events you specified for exclusion in the view definition.

Figure 5 shows the user interface for view creation and modification. The view range is a lower and upper bound of time or events, which you select or adjust. For event adjustments, the view manager searches for the named event forward or backward from the current lower or upper bound. You make time adjustments either by entering a new time directly, or by using the scroll bar. While it maintains its normal scrolling functions, which let users move a time window across the trace, the scroll bar can also be extended or contracted to change the time-window size.

For event filtering, you toggle event names in the event filter list. You can toggle them individually or collectively, using the string pattern-matching capabilities. You change the effect of the pattern matching using the Exclude and Include buttons.

Traceview performs event filtering based on the view definition when constructing a virtual trace. Removing the events that occur before or after the viewing range is trivial. Filtering out individual unwanted events is a bit more complicated. The view manager removes an event from the trace if the event's To or From state designator matches an event name specified for exclusion. The easiest way to understand the algorithm for event filtering is to see it as removing one event at a time from the trace. At each iteration, the algorithm creates a new virtual trace with all events to or from the state in question removed. The data fields are updated appropriately. When it has removed all the specified events, the final virtual trace is the result.

Because trace files can be large, the cost of reading raw traces entirely into memory can be prohibitive. However, always reading trace events from secondary storage can slow the system. Our approach in Traceview is to cache trace events in memory buffers. We chose to cache virtual traces instead of raw traces because virtual-trace range selection and event filtering suppress events of no concern in the display. However, Traceview does virtual-trace caching only for views, not for each derivative view display.

Traceview implements virtual-trace caching by assigning each defined view a cache buffer, the size of which you control through a Traceview application resource. (In the future, we will add interactive buffer-size control to the view window.) At any time, some consecutive portion of the virtual trace is cached. The time stamp of the first cached event and the time stamp of the last cached event delimit the portion. If a display requests a virtual-trace event within the cache, the view manager fetches the event from the cache, getting the benefits of fast memory access. However, if a display requests a virtual-trace

event outside the cache, the view manager flushes the cache and then refills it from the virtual-trace stream derived from the original trace on secondary storage, starting with the requested event.

Virtual-trace caching minimizes Traceview's total memory requirements while offering fast response time during display interaction. Speed depends on several parameters: the cache-buffer size, the view-range selection, the extent of event filtering, and the location of the virtual-trace references during display update. Further experience with Traceview, particularly with large trace files, will help us determine the efficacy of virtual-trace caching.

**Display management.** The display manager graphically presents the virtual trace constructed by the view manager. The display manager lets you select from two available display methods and creates a display window showing the data. The display manager controls the right window in Figure 3, which presents a list of existing displays for the selected view. You can reopen, destroy, and create displays.

The display manager does not actually display the data; rather it invokes a display method to display a virtual trace. Traceview does not dictate a display method. Although two display types are statically linked with Traceview, you can incorporate any display method that can interpret a Traceview virtual trace.

## DISPLAY METHODS

The display type menu lets you select either a Gantt display or a rate display. Both use the Gantt chart widget. Gantt charts are line-plot representations of time-sequenced performance data. Traceview uses a Gantt chart widget we developed expressly for displaying trace data.

**Gantt chart widget.** The Gantt chart widget provides horizontal and vertical axes, axes labels, data display, density bars, and data averaging. In most cases, the number of points displayed in a Gantt chart greatly exceeds the pixel width of the $x$ axis of the Gantt chart. In our experience, the ratio of data points to pixels commonly exceeds 10:1. The Gantt chart widget offers two solutions to this data-density problem: density bars and average curves.

The display of a density bar on a chart is optional. A density bar is a band of color displayed above the Gantt's data-display area. For density bars, Traceview lets you select a color map, which assigns integer ranges to colors. A density bar can represent either value density or point density. In a value-density bar, the color at pixel $p$ represents the average of all the data points displayed at pixel $p$ on the $x$ axis. In a point-density bar, the color at pixel $p$ represents the number of data points represented at pixel $p$ on the $x$ axis.

In addition to a density bar and the graphical data display, the Gantt chart widget optionally displays an average curve. The average curve overlays the data display and is computed by taking the average over a fixed interval of points as the interval slides along the $x$ axis. The average data value for point $p$ on the $x$ axis is calculated using all the data points from the interval when centered at $p$. If a point on the $x$ axis has no actual data points associated with it, the Gantt chart widget assumes it has the same average value as the most recent point preceding it.

Although two display types are statically linked with Traceview, you can incorporate any display method that can interpret a Traceview virtual trace.

**Display shell.** Both display methods bundle Gantt charts in a display shell that synchronizes many Gantt charts in one window. The display shell also provides the user interface where you control the behavior and appearance of the individual Gantt charts.

All the Gantt charts in a display shell derive from the same virtual trace, and the $x$ axis for all the charts is identical. Therefore, the display shell stacks Gantt charts horizontally and aligns them so a vertical line across the display shell intersects all its charts at precisely the same point on the $x$ axis. Because of this vertical stacking, the density and distribution of data points on the $x$ axis are identical for all the Gantt charts in a given display shell.

The display shell lets you control the individual Gantt charts. From the display shell you can add charts to or delete charts from the display, turn density bars on or off, turn averaging on or off, and produce a Postscript version of the chart.

The display shell also lets you manipulate all the Gantt charts together by adjusting the averaging interval and by zooming in on a chart region. You select the averaging interval by manipulating a slider to choose a percentage of the width of the chart's $x$ axis. To zoom in on a chart portion, you use the mouse to select a region. Then you can zoom in on that region in all the charts. You can undo zooming stepwise or all at once.

**Using Gantt charts.** Traceview uses the Gantt chart widget with the display shell to present two kinds of displays to the user: displays based on state transitions, which we call simply *Gantt displays*, and displays of the number of times a state is entered, or *rates displays*.

For Gantt displays, the chart's $x$ axis represents time. The $y$ axis varies from chart to chart. For traces that contain no data fields, the display manager shows only a single chart. Here, the $y$ axis represents states. A square wave shows when a state is entered and when it is exited. If the trace also includes data fields, then the display shell includes a Gantt chart for each data field. For these, the $y$ axis represents the data field values. Because Traceview traces only have data recorded at state transitions, all the charts have the same $x$ axis and the same data-point distribution.

In the rates display, you view metric data associated with a trace state. For the selected state, you can have a chart for each data field defined for the trace. The

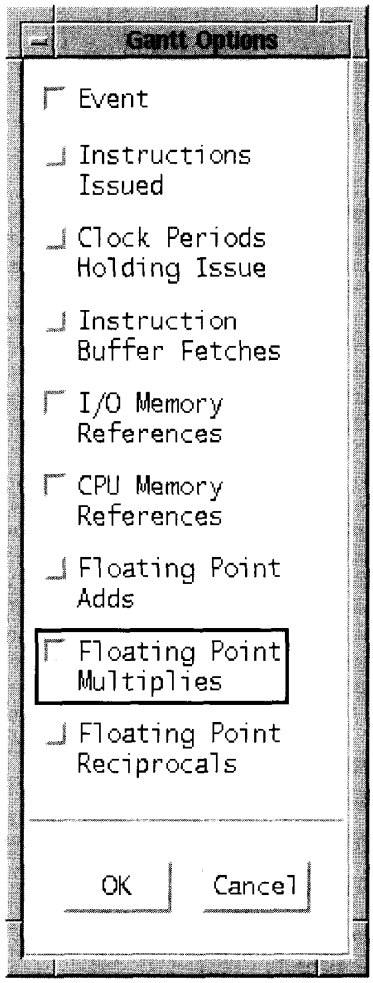


*Figure 6. Gantts display options.*

display manager also provides an additional chart giving the summation of all the metrics available.

For the rates display to be applicable, the trace must include data fields. In the rates display, the *x* axis represents ordinal instances of the state being entered. The *y* axis represents the individual data field values. Instead of the more familiar square wave usually used in Gantt charts, the display manager causes the Gantt chart widget simply to display the points and connect them.

**Miscellaneous features.** You can use the mouse to place a horizontal line across the Gantt chart plotting state transitions and find out the name of the state represented by that point on the *y* axis. Conversely, you can bring up a list of all states in the virtual trace, select one, and have the corresponding horizontal line displayed across the Gantt chart.

From within the display shell, you can dump the details of a virtual trace to an ASCII file. Traceview generates two types of files. The first contains the details of each event record in the virtual trace, including time stamps and all the other fields of the event record. The second contains the details of how Traceview constructed the virtual trace from the raw trace. If Traceview excludes routines from the virtual trace, the most relevant data in this file is how Traceview combined or ignored event records and their data fields to create the virtual trace. We use the second type of file for diagnostics in virtual-trace generation.

## TRACEVIEW APPLICATIONS

A general-purpose trace-visualization tool must be effective across a variety of applications. Using the tool's standard features, you should be able to conveniently process and visualize traces from real or simulated systems. The visualization should help you gain insight into important performance phenomena difficult to observe otherwise. The tool should let you input trace data from different performance levels (hardware, system software, and application software) with different data appearing together in the trace. Finally, a trace-visualization environment must let you visualize and compare multiple traces simultaneously.

To see whether Traceview met these practical objectives, we used it to visualize traces encountered in our performance evaluation projects at the Center for Supercomputing Research and Development at the University of Illinois at Urbana-Champaign. One project required Traceview to visualize program-event transitions and associated hardware performance information from applications programs running on Cray supercomputers. Another project was to determine via simulation the maximum parallelism within application executions. In this project, Traceview shows time-dependent levels of parallelism and corresponding system performance metrics. We elaborate on these two projects later. Another project not discussed here was the visualization of register- and functional-unit usage in traces from an instruction-level simulation of a single Cray X-MP processor.

We conclude from our validation tests that Traceview is effective across different trace-visualization applications and requires little trace input modification, except for conversion to the standard Traceview trace format. Although Traceview's display methods are currently limited, the Gantt and rates displays offer a robust visualization approach, particularly for comparing different performance data within a single trace or across multiple traces.

We observed shortcomings in those trace data analyses that required semantic interpretation. In the future, we could add limited semantic capabilities to Traceview for applications that require, for the most part, semantically independent trace analysis, but need some semantic trace interpretation. However, we will continue to emphasize trace-visualization applications in which the user provides the semantic context.

**Viewing Cray applications.** You can capture detailed data about the performance behavior of an application's execution only by tracing important events and sampling relevant machine performance metrics. One tracing system for Cray supercomputers[7] measures program-event transitions and stores information about machine performance accessed from the Cray hardware-performance monitor.[8] There are four hardware performance monitor groups, covering different classes of hardware performance, but only one of the four groups is accessible at a time.

Tracing application execution can generate large amounts of trace data quickly, and analyzing and presenting the data manually can be arduous. We have applied Traceview to the analysis and display of Cray application traces. Here we describe a sample Traceview session, where we viewed the trace from a vector execution of the Perfect benchmark code FLO52 (a multigrid fluid-dynamics computation).[9]

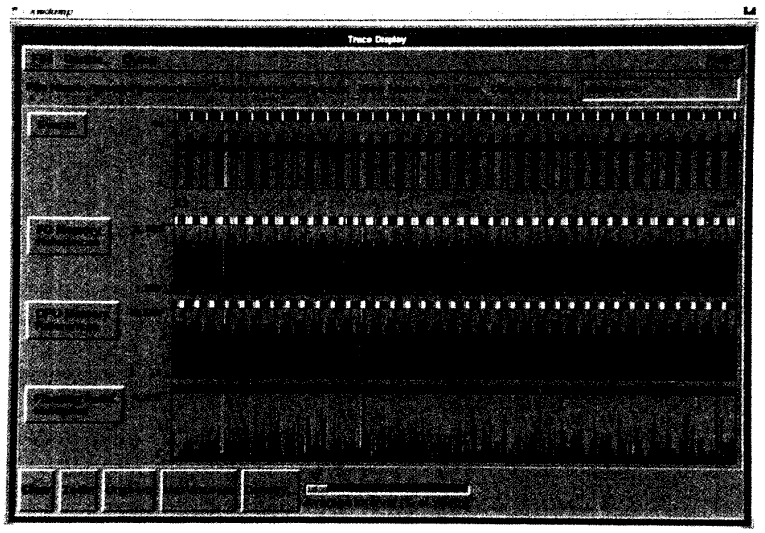


***Figure** 7. Gantts display of the transitions between FLO52 computational blocks during execution and the corresponding hardware behavior. The Events charts shows all entry and exit transitions: Each vertical line represents a routine transition; each horizontal line time spent in the current routine. The colored point-density bar ranges from black for least dense, through red, orange, and yellow, and finally to white for most dense. Below the Events chart are hardware-performance metrics, computed as rates over the period between successive state transitions. In this case, the rates are in millions per second. From these displays, you can see how hardware performance changes between routines and correlate performance across the different hardware metrics. The figure shows that the cyclic nature of the routine transitions in the displayed region of the FLO52 computation is reflected in repetitive hardware behavior. This indicates relatively stable hardware performance within and between the successive periods. The value-density bars show the average performance value at the screen resolution.*

The session display in Figure 3 is a snapshot of the session configuration used for the FLO52 traces. The Files window lists the four open trace files from different vector executions of the FLO52 program. The file ftrace contains an events-only trace, while the other traces include hardware-performance monitor values for each event in the trace. The ptrace file contains the group 0 counter values. The files trace02 and trace03 contain the group 2 and group 3 traces, respectively. Figure 4 shows the summary trace information for ptrace.

Because the ptrace trace contains both event data and hardware-performance monitor data, you must be able to view event transitions and changes in hardware performance simultaneously. Then you can analyze different computation regions to see their effects on hardware performance. Figure 6 shows the Gantt Options menu, which lets you choose interactively the data to be shown. Figure 7 shows the Gantts display of the transitions between FLO52 computational blocks during execution and the corresponding hardware behavior.

Because the view definition for Figure 7 selects all routines for viewing, all entry and exit transitions between the FLO52 routines appear in the Events chart. All routines are numbered and appear at their corresponding number level on the *y* axis in the Events chart. You can bring up the routine-to-number mapping in another window for review. Each vertical line in the display represents a routine transition, and each horizontal line indicates the time spent in the current routine. Thus you can identify the current routine at any point in the display and observe the relative pattern of routine occurrence and the differences in routine execution time. For FLO52, the Events chart shows the cyclic nature of the multigrid computation during one major phase of the execution.

Figure 7 also shows the values of the chosen hardware-performance metrics corresponding to each routine transition. The displayed values are computed as rates over the period between successive state transitions. In this case, the rates are in millions per second. From these displays, you can see how hardware performance changes between routines. You can also correlate performance across the different hardware metrics. The figure shows that the cyclic nature of the routine transitions in the displayed region of the FLO52 computation is reflected in repetitive hardware behavior. This indicates relatively stable hardware performance within and between the successive periods.

A problem with displaying a large number of events is screen resolution. The screen reproduced in Figure 7 displays approximately 11,000 events in 900 pixels. The point-density bar in the Events display highlights areas where many events are presented. In this case, the color map ranges from black, through red, orange, and yellow, and finally to white. Black is the least dense, white the most. The value-density bars in the various performance-metric displays show the average of performance values at the screen resolution. Zooming in on areas with high point density provides additional details about event transition and hardware performance.

Figure 8 shows a zoomed-in Gantts display for the FLO52 trace. Although there is still an event-density problem, you can discern some individual event transitions. This lets you observe and quantify the high variability in discrete hardware performance transitions during the FLO52 computation. Clearly, the CPU Memory References metric is not uniform across routines and shows significant changes in the memory reference rate. We also observed this for the Total Floating Point Operations metric (not shown).

You use the average curve, drawn over the CPU Memory References metric chart, to contrast the discrete performance behavior with average performance. The CPU Memory References metric is a good example of how Traceview can simultaneously display performance data analyzed over different time intervals. Viewing detailed hardware data localizes the performance behavior to particular routines, while averaged hardware data curves show aggregate performance over time.

You use Traceview's rates display to observe the performance differences between successive invocations of a single routine. Figure 9 shows four performance

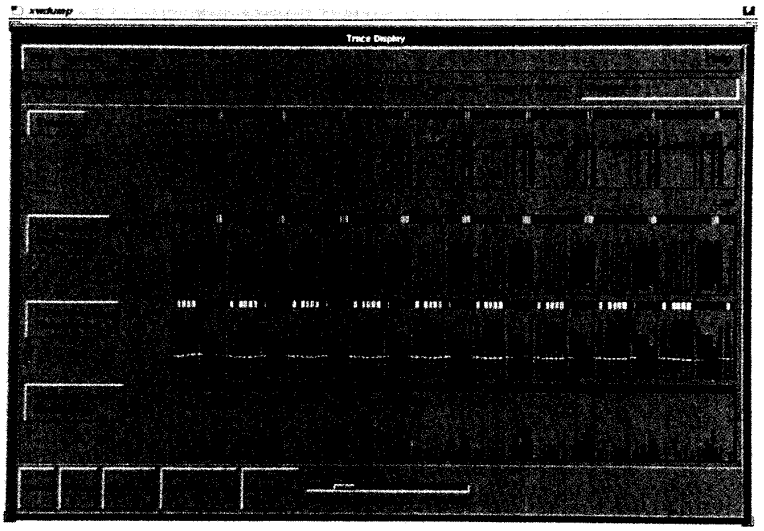


*Figure 8. Zoomed FLO52 Gantts display which lets you discern some individual event transitions. The CPU Memory References metric is not uniform across routines and shows significant changes in the memory reference rate. We also observed this for the Total Floating Point Operations metric (not shown).*

metrics for each invocation of the routine Psmoo, a solution-approximation smoothing routine in the multigrid FLO52 code. A transition in the display reflects a separate invocation in a time-ordered sequence of Psmoo calls. Because no resolution problem exists, you can quickly observe the cyclic nature of Psmoo performance in the view region. All Floating Point Operations and CPU Memory References are positively correlated, but they are negatively correlated with I/O Memory References. Floating Point Multiplies appear to be uncorrelated. The invocation sequence in a single period reflects the FLO52 computation's multigrid nature. As the Psmoo routine is called on successively coarser grids, the vector operations increase in overhead, resulting in decreases in delivered floating-point performance and memory-reference rate.

The main benefit of the rates display for the FLO52 application study is its ability to focus on a particular routine and display performance data for only that routine. All the user-interaction capabilities of the Gantts display are accessible in

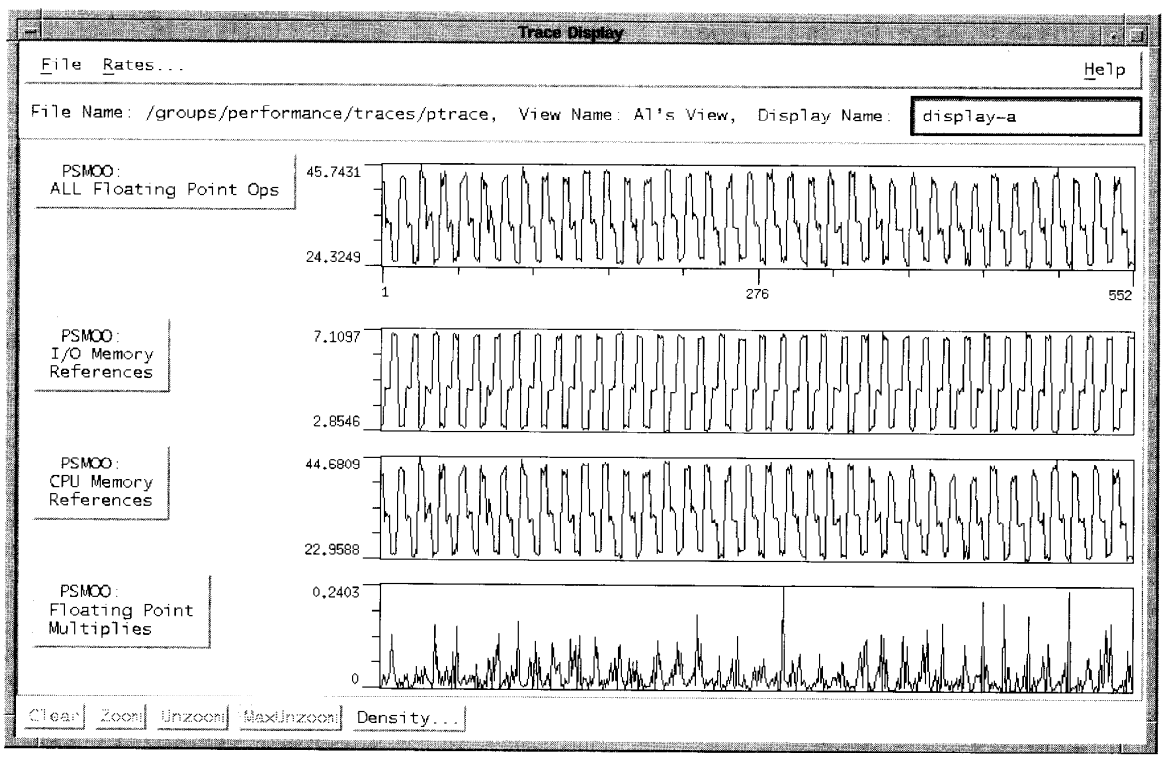


*Figure 9. Four performance metrics for each invocation of the routine Psmoo, a solution-approximation smoothing routine in the multigrid FLO52 code. A transition in the display reflects a separate invocation in a time-ordered sequence of Psmoo calls. This rates display focuses on a particular routine and displays performance data for only that routine.*

the rates display.

We continue to use Traceview to study different versions of Cray applications, including FLO52. Because Traceview lets us open multiple trace files simultaneously, we can view traces from different executions on the screen at the same time. This is important in Cray work because we must execute the programs several times to capture traces for all four hardware-performance monitor groups. Also, we want to compare the FLO52 traces of vector execution with those from scalar and concurrent executions to better understand the relative performance benefits of different compiler optimizations.

**Viewing maximum parallelism.** When evaluating the performance of a parallel computation, you want to know the maximum levels of parallelism that could be achieved during execution. Although in practice the instantaneous parallelism will be limited by the total processors available on the machine, a notion of peak parallelism can help you understand execution efficiency.

Maxpar[10] is a simulator for extracting a program's maximum theoretically attainable parallelism. It works by maintaining a set of shadow variables for each actual variable. The shadow variable records the time when the variable's current value is valid. Each operation in the original program, in addition to computing a change in the variable's value, also computes the time when that value is first available.

From this information, Maxpar can calculate the number of operations that can be simultaneously computed. From these maximum parallelism levels, Maxpar can derive other operational statistics such as the number of CPU memory references and floating-point operations.

Maxpar generates extensive tracing information, roughly comparable to that produced by Cray's hardware-performance monitoring software. It generates event records on each routine entry and exit, along with information on when the routine could have first been entered. In addition, performance profiling records keep track of when each instruction and arithmetic operation was ready to execute. A set of filters adapts this information to a format suitable for Traceview.

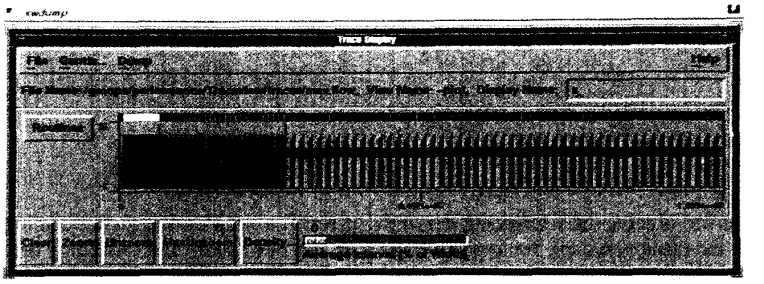

*Figure 10. Gantts display of the Events chart of the entire FLO52 computation as analyzed by Maxpar. The chart clearly shows the three major phases of the FLO52 computation. However, unlike scalar and vector Cray executions, multiple routines can be active at the same time.*

Once they are formatted, you can view the Maxpar profiles side by side with Cray traces of the same program. This lets you compare potential peak parallelism, as generated by Maxpar, with the Cray's attained performance. Figure 10 shows a Traceview Gantts display of the Events chart of the entire FLO52 computation analyzed by Maxpar. The chart clearly shows the three major phases of the FLO52 computation. However, unlike scalar and vector Cray executions, multiple routines can be active at the same time.

Figure 11 shows a Gantts display for Maxpar performance data generated from FLO52. The Instructions Executed Gantt chart indicates the number of parallel instructions executed. In comparison with a real traced execution, the Maxpar traces reveal the same phased, periodic behavior of the FLO52 computation. However, the levels of parallelism are significantly more enhanced and variable from phase to phase. The Instructions Executed levels in the measured Cray traces are limited by the number of processors available and the vector register lengths.

As with a Cray execution trace, it is interesting in the Maxpar case to observe how instruction execution affects machine performance. You can use the Store Ops

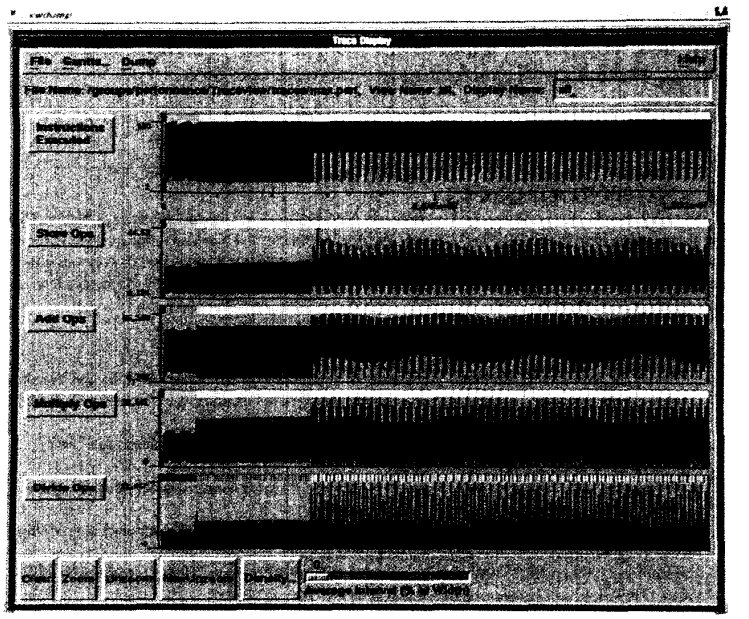

*Figure 11. Gantts display for Maxpar performance data generated from FLO52 metrics. Maxpar traces reveal the same phased, periodic behavior of the FLO52 computation, but the levels of parallelism are significantly more enhanced and variable from phase to phase. It is interesting in the Maxpar case to observe how instruction execution affects machine performance. The Store Ops chart shows the increase in simultatenous memory references, which is expected. However, performance variability also increases. Similarly, the Add Ops chart shows an expected larger number of floating-point operations at higher levels of parallelism.*

chart to observe the referencing demand on the memory system. You expect a large number of parallel instructions to generate a large number of simultaneous memory references, and the Store Ops chart shows this quite clearly. However, the performance variability also increases. Similarly, you expect a larger number of floating-point operations at higher levels of parallelism. The Add Ops chart shows this pattern. You can interpret this chart as showing maximum floating-point add operations.

Many Traceview features are useful for the Maxpar application. For example, you can see the average performance on the graphs at various averaging intervals and evaluate how well the FLO52 computation as a whole benefits from parallelism. Traceview's cross-trace comparison capabilities also let you evaluate different optimized Cray executions against the maximum parallelism cases to determine the optimizations that contribute most effectively to performance.

The Traceview model does not apply to all trace-visualization applications. However, we think it will be useful in most cases. Traceview's primary shortcoming is its lack of semantic understanding of the trace events and data. In our future research, we must determine whether adding semantic knowledge to increase analysis capabilities will lessen the tool's general applicability. Perhaps we can identify common trace types and incorporate basic events and analysis capabilities derived from these types into Traceview. Or we could make the tool's architecture more open to let users write their own special-purpose analysis and display components and integrate them with the tool's general trace-management and view-specification features.

We intend to add the capability to visualize resource usage where the data associated with an event defines changes in resource state rather than a performance metric. For this, a color-coded PERT (performance-evaluation research task) display might be better than a Gantt line-plot display. Such functionality would be well suited to trace visualizations of CPU function-unit usage, processor use, memory-module references, and switching-network operation. ♦


## ACKNOWLEDGMENTS

We thank Sam Ho for providing the Maxpar traces. Malony's work was supported in part by National Science Foundation grants NSF MIP-88-07775 and NSF ASC 84-04556, and National Aeronautics and Space Administration Ames Research Center grant NCC-2-559. Hammerslag's and Jablonowski's work was supported in part by US Air Force Office of Scientific Research grant AFSOR 90-0044 and US Energy Dept. grant DE-FG02-85ER25001.


## REFERENCES


1. A. Malony, D. Reed, and D. Rudolph, "Integrating Performance Data Collection, Analysis, and Visualization," in *Parallel Computer Systems: Performance Instrumentation and Visualization*, M. Simmons, R. Koskela, and I. Bucher, eds., ACM, New York, 1990.
2. D. Reed and D. Rudolph, "The Intel iPSC/2: An Approach to Performance Instrumentation," *Int'l J. High-Speed Computing*, Dec. 1990, pp. 517-542.
3. B. Miller et al., "IPS-2: The Second Generation of a Parallel Program Measurement System," Tech. Report CS-783, Univ. Wisconsin, Madison, Wis., 1988.
4. T. LeBlanc, J. Mellor-Crummey, and R. Fowler, "Analyzing Parallel Program Executions Using Multiple Views," *J. Parallel and Distributed Computing*, June 1990, pp. 203-217.
5. M. Heath and J. Etheridge, "Visualizing Performance of Parallel Programs," Tech. Report ORNL/TM-11813, Oak Ridge Nat'l Laboratory, Oak Ridge, Tenn., 1991.
6. T. Lehr et al., "Visualizing Performance Debugging," *Computer*, Oct. 1989, pp. 38-51.
7. A. Malony, J. Larson, and D. Reed, "Tracing Application Program Execution on the Cray X-MP and Cray 2," *Proc. 1990 Supercomputing Conf.*, CS Press, Los Alamitos, Calif., 1990, pp. 60-73.
8. J. Larson, "Cray X-MP Hardware Performance Monitor," *Cray Channels*, Winter 1986, pp. 18-19.
9. M. Berry, "The Perfect Club Benchmarks: Effective Performance Evaluation of Supercomputers," *Int'l J. Supercomputer Applications*, Fall 1989, pp. 5-40.
10. D. Chen, *Maxpar: An Execution Drive Simulator for Studying Parallel Systems*, master's thesis, Univ. of Illinois at Urbana-Champaign, Urbana, Ill., 1990.



**Allen D. Malony** is an assistant professor in the Computer and Information Science Department at the University of Oregon. While working on Traceview, he was a senior software engineer at the Center for Supercomputing Research and Development at the University of Illinois at Urbana-Champaign. His research interests are performance evaluation, multiprocessor architectures, and parallel programming environments.

Malony received a BS and an MS in computer science from the University of California at Los Angeles and a PhD in computer science from the University of Illinois at Urbana-Champaign. He is a member of the IEEE Computer Society.



**David H. Hammerslag** is a software engineer at the Center for Supercomputing Research and Development at the University of Illinois at Urbana-Champaign. His research interests are software engineering, programming environments and tools, and programming languages.

Hammerslag received his BS, MS, and PhD from the University of Illinois. He is a member of the IEEE Computer Society.



**David J. Jablonowski** is a senior software engineer at the Center for Supercomputing Research and Development, where he specializes in integrated environments and user-interface design.

Jablonowski received a BS in computer science from the University of Wisconsin at Eau Claire and an MS in computer science from Boston University.

Address questions about this article to Hammerslag, Center for Supercomputing Research and Development, University of Illinois, Urbana, IL 61801; Internet hammer@csrd.uiuc.edu.