Displaying Trace Files

Stephen G. Eick† and *Paul J. Lucas*
AT&T Bell Laboratories

29 June 1995

ABSTRACT

Computers generate trace files containing reports on system performance, status, and faults. To analyze these trace files more efficiently, we have developed a graphical technique embodied in an interactive system for displaying large trace files. Our system uses abstraction, color, aggregation, filtering, interaction, and a drill-down capability to find patterns among the reports. We apply our system and technique to analyze command accounting trace files from a Unix compute server, showing what commands were executed, by which users, when, and how long the commands ran. We identify resource intensive commands, sequences of commands initiated by a compilations, and commands run with super-user permissions.

1. Introduction

Trace files contain streams of time-stamped, typed messages or reports and are generated by many computer systems. Associated with each report is an event time, possibly a duration, and one or more report characteristics. Examples include transactions reports from a financial system, electronic mail traces, facility alarms from a transmission monitoring system, project status reports from a management system, packet traces through a routerm and command accounting traces from a multi-user computer system. These reports are often free format and may be in a fixed file or may be generated continuously. Trace files containing reports on system performance, status, hardware errors, software faults, and

† Correspondence Contact: AT&T Bell Laboratories - Room 1G–351, 1000 East Warrenville Road, Naperville, IL 60566-7013, e-mail: eick@research.att.com

security violations are indispensable to maintain successful system operation. In many cases they are the only record of the system's activities leading up to a problem. The files are analyzed and the reports are used by system managers for trouble shooting, system debugging, pro-active maintenance, performance tuning, and system surveillance.

Analyzing a large trace file containing thousands to tens of thousands of reports is a daunting task. Many of the reports are unimportant and create clutter and that obscures the important reports, preventing significant problems from being detected. The analyst must examine each report individually to determine which are important and which may be ignored. Detecting patterns in such a large trace is difficult as is understanding overall patterns. Since the reports are concatenated together in a trace stream, temporal patterns are particularly difficult to detect. The reports generated during five minutes of system operations may span a few hundred lines during a "quiet" period or can span a few thousand lines during a "noisy" period. In practice, the most common way trace files are analyzed is with a text editor, with string searches, or with string matching commands such as `grep`. This approach can be frustrating because it is easy to get lost within the trace file, lose context, and miss important clues.

To aid this task, we have developed a visualization technique embodied in a system called *SeeLog*.™ It uses interactive graphics, data-visualization, and direct-manipulation to browse trace files while maintaining an intuitive and friendly user interface. *SeeLog* borrows many of the display and interaction techniques from *SeeSoft*[1] and its application toward analyzing trace files.[2] *SeeLog* runs on workstations under the X Window System[3] and we are currently porting it to Microsoft Windows. A design goal of *SeeLog* is to provide an overview of a trace file showing temporal patterns, filtering mechanisms to highlight problems and to discover correlations between the reports, and a drill-down capability to provide detailed information.

Our original interest in trace files comes from analyzing those generated by the *5ESS*® during development and from supporting operational switches in the field. The *5ESS* is a

distributed, fault-tolerant, telecommunications switch developed by AT&T. Its software, composed of several million lines of code written by a few thousand developers, is continually being modified for customers world-wide. For each release of the software, many incremental versions are built and tested. At our site there are approximately 25,000 testing sessions per year, and for each there is a trace file that must be analyzed. Some of the most complicated trace files may contain tens to hundreds of thousands of report comprising hundreds of thousands to millions of lines of text. Analyzing these traces is extraordinary difficult and time consuming.

To illustrate our technique, we use the Unix System V command accounting facility. This trace file contains a report for each command executed and is automatically generated as part of standard operations. It contains a detailed history of the machine's activity and it is used by system administrators for performance tuning, security monitoring, and could be used for usage billing. We find it particularly interesting because, by studying the traces from one of our own machines, we gain insight into how we in a research department use computing resources, and have discovered commonly executed sequences of commands that could be used for human factors studies.[4] [5] By analyzing this data, we have gained some interesting insights into our work patterns.

2. Background and Previous Work

Because analyzing trace files is tedious, it has not received the academic attention it deserves. We think this is unfortunate because of the large numbers of trace files generated by computers and the amount of time spent in analyzing them. Other researchers have recognized this important problem and are focusing on trace files generated by parallel systems, perhaps because debugging parallel programs is so difficult. Two state-of-the-art examples include ParaGraph,[6] a general-purpose animation system for parallel programs, and Pablo,[7] a complete analysis environment with sophisticated three-dimensional immersive data visualizations. Both of these systems are well-suited to visualize the message traffic between processors and have parallel measurement and instrumentation

capability, providing a rich source of data to visualize.

Our focus is on a special class of trace files containing streams of time-stamped, typed events. Our motivating example involves the trace files produced by semi-weekly lab stability runs of the 5ESS. These trace files can be very large, a trace file from a 15 hour lab stability run might contain over 50,000 time-stamped events comprising 100,000 lines of text. Because of the complexity of the 5ESS and the domain knowledge required to understand a 5ESS trace file, we will illustrate our techniques using a standard the command-accounting files commonly created by Unix systems. This file also contains time-stamped list of every Unix comand execution.

Common practice is to use standard text editors such as `vi` or `emacs` to browse trace files looking for patterns. However, browsing in this way obscures the inherent nature of time-stamped trace files. The text editor can only view a small portion of a large trace and emphasizes the spatial separation of interesting reports (i.e., the number of lines between them) but this spatial separation is arbitrary and depends on the noise in the trace file. As we pointed out previously, five-minutes' worth of trace file may contain anywhere from a hundred lines to a few thousand, depending on how many reports were generated. When using a text editor or a printout it is nearly impossible to detect patterns of reports.

Sophisticated organizations will develop tools to calculate report summaries and perform statistical analyses on the errors in their trace files. It is difficult, however, to relate statistical trends to actual reports, and, in particular, to the software. This approach is ad-hoc and depends on the structure of trace file under study. For each new class of trace files, new statistical techniques have to be developed.

Eick, Nelson, and Schmidt use the *SeeSoft* text visualization system to display large log files.[2] They divide the trace file into hours and represent the lines of text in reduced form, as horizontal rows stacked in vertical columns, one column for each hour. The lines are color-coded according to the message type. Unfortunately, this approach, while novel and useful for some classes of trace files, cannot handle trace files with more than 100,000 lines. It also

suffers from the same spatial separation problem as does browsing with a text editor: equal increments in text do not correspond to equal increments in time. This problem complicates detecting time-oriented faults.

Another approach for analyzing trace files employs expert systems, such as the 5XPRESS,[8] that searches trace files for patterns. *SeeLog* is complementary to the expert systems approach. By presenting the error reports visually, we can rely on the brain's ability to detect patterns, and solve difficult computational problems by perception. Once the patterns are known, we can develop an expert system to search automatically for them.

3. Visualization and Interaction Techniques

Our trace file analysis paradigm involves two steps: parsing and visualization.

3.1 Trace File Parsing

Parsing a complicated trace file involves lexicographically scanning it to note the times, types, and locations of all reports. This step can be done using tools like `grep`, AWK,[9] Perl,[10] or even a C program. The data from the scanning are placed in a table that is the input to *SeeLog*.

Parsing may also involve some filtering to select only the "interesting" reports, for example, those that signify commands of interest in the Unix accounting example. Filtering eliminates some of the many noise reports that do not contribute to the discovery of patterns and correlations, and that obscure those that do. Such reports are still useful, however, for doing more detailed analysis.

3.2 Display Method

To create a visual display of a trace file, the reports are arranged chronologically and grouped by type. Each report is then represented as an angled "tick mark" on a grid with time running along the x-axis and report type along the y-axis. The report types listed along the y-axis may be placed into bands of related types value. The result is a pattern of

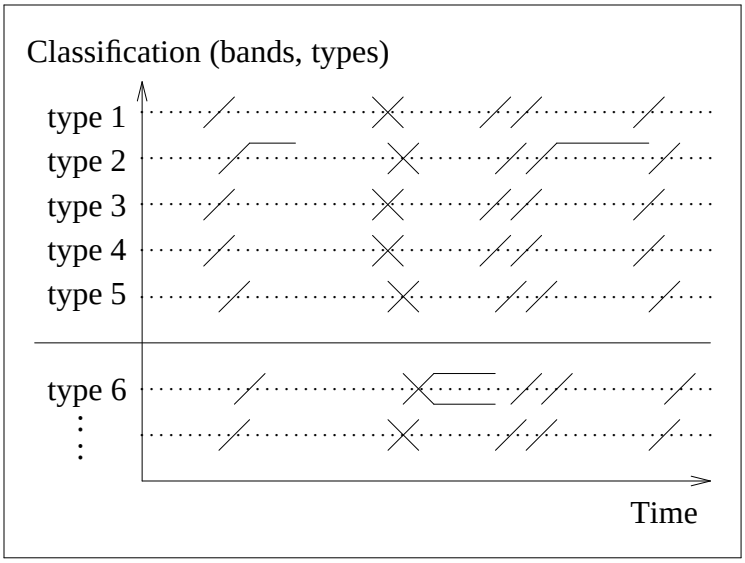

Figure 1. Log File Display

Each tick mark represents one report and is positioned on a grid chronologically and grouped by type. The *x*-axis encodes time and the *y*-axis type.

horizontal bands, each containing a number related of rows, with ticks indicating occurrences. (See Figure 1.)

Within each band, there are rows for the distinct values of each type. The type name is printed at the left side of the display and the type value is printed next to its corresponding row. The rows may be sorted in decreasing tick mark frequency or in alphabetical order.

In most datasets, there are several dimensions of type information. For example, in the command accounting dataset the type information includes the user-id, number of characters transferred, and process size. There are three methods to encode different dimension of type information: rows (primary method), tick mark color,[1] and tick mark angle. The color and angle of each tick mark may encode different type information, but often redundantly encode the same type information.

1. In a black and white copy of this paper, color is rendered as a gray level. Additionally, the figures here are negative images; the real screen display has light graphics on a black background.

3.3 Color And Angle

Using both the color and the angle to encode type information is complementary. First, for gray-scale monitors, it is hard to perceive the gray-level of the tick marks. Second, when there are many values of an attribute, colors of adjacent hues can be too close to discriminate. We use a predefined set of six angles that differ sufficiently for easy discrimination where each value of an attribute is sequentially assigned to the next angle with wrapping. Thus, the values that are close in hue are far apart in angle, permitting easy discrimination.

Angle-coding also addresses another difficult problem: overplotting. When many reports occur at nearly the same instant in time, their corresponding tick marks are plotted either very close or directly on top of each other. This results in a difficult-to-interpret display where bursts of reports are encoded with what appears to be a single symbol. Varying the tick mark angles neatly solves the overplotting problem: multiple tick marks appear prominently as "starbursts." These are shown in Figure 1 by tick marks like ×.

3.4 Durations

Some reports correspond to events with significant durations. A tick mark by itself encodes the starting time of a report. This works well for short duration events, such as executing simple commands in the Unix command accounting trace file. We encode longer events by adding "tails" to their tick marks, extending to the right showing their duration. These are shown in Figure 1 by tick marks like / ‾‾.

3.5 Important Patterns

When analyzing trace files, certain visual characteristics of the display are often interesting:

1. A "starburst" of different overplotted angles indicates that for one value of the row type, there are several reports with distinct values of the angle type occurring almost simultaneously. This is often an indication of a burst of related reports.

2. A “wave” is a vertical pattern of tick marks corresponding to different types of reports occurring at nearly the same x-position. These reports occurred at nearly the same time and are often related.

3. An “alternating sequence” of tick marks in different rows indicates that different types of reports may be related.

4. A “repetitive vertical pattern” of tick marks occurring consistently throughout the display indicates a periodic sequence of events in the trace file.

4. An Example—Unix Command Accounting

Unix computer systems maintain accounting traces with a record for each command executed, either from the command-line or from a shell script. On our department’s compute server,[2] the accounting trace file (`/usr/adm/pacct`) contains the name of the command, the user-id under which it was executed, whether it was executed with super-user permissions, its starting time, real-time, CPU-time, characters transferred for I/O, mean process size, and a few other pieces of information. This trace file contains a complete history of the command executions on the machine. It is used file for performance tuning, load balancing, security monitoring, and to plan for future hardware upgrades.

When analyzing a computer trace file, a system manager would like to answer these seven questions:

- *What is the usage pattern on the machine?*
- *Which commands are executed frequently?*
- *When does it have unused capacity?*
- *Which user’s execute the most commands?*

2. A Silicon Graphics Challenge/XL machine running Unix System V.4 currently configured with four processors.

- *Are certain commands particularly resource-intensive?*
- *How does the CPU time get used?*
- *Who is running commands with super-user permissions?*

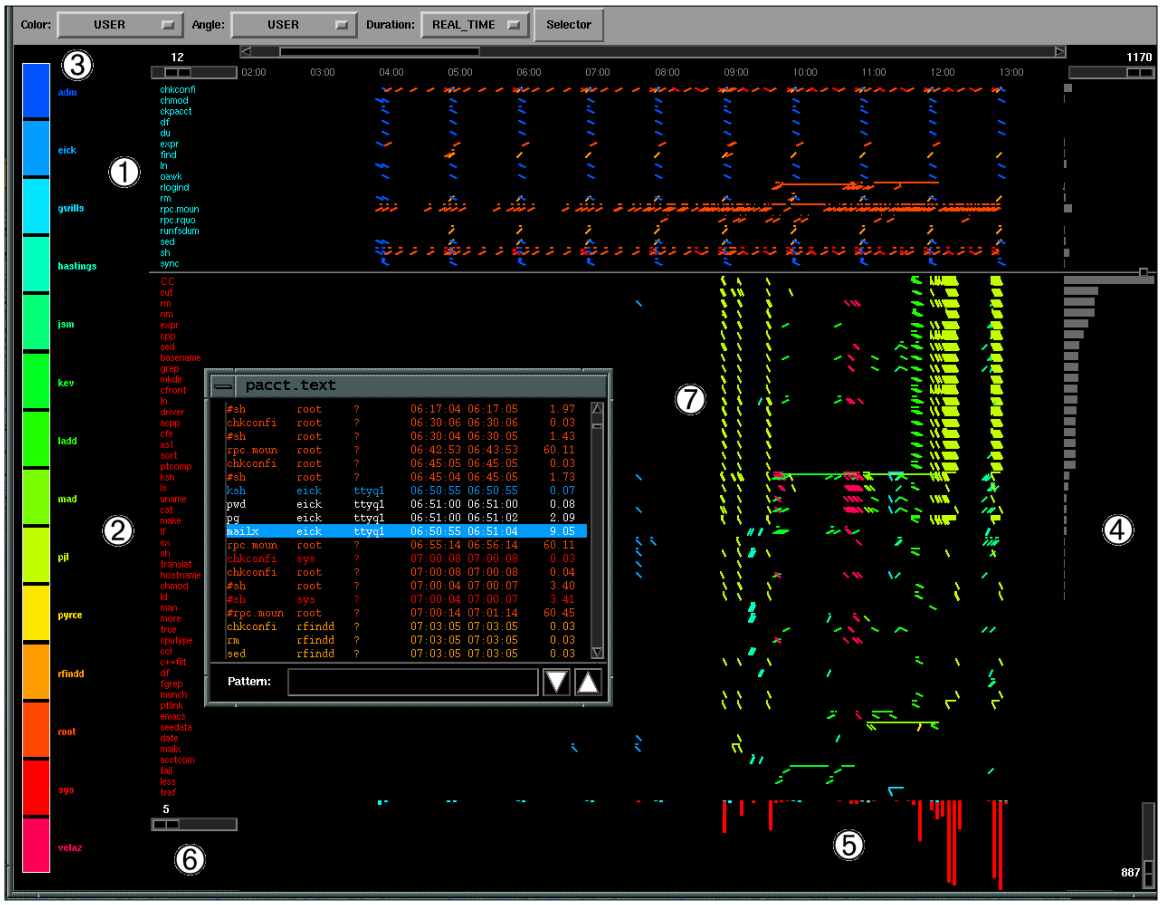

Figure 2. Who did what: Coded by user-id

Each tick mark represents one Unix command or shell script that executed during a ten hour period on our compute server. There is one row for each command and a tick mark representing an execution. The tick marks are positioned chronologically and color- and angle-coded to show the user-id executing that command.

4.1 Coding by User-ID

The display of the trace file in Figure 2 shows ten hours worth of data from 4:00 to 13:59 from our department's compute server. During that period, 6179 command accounting reports were generated. (We have looked at command traces containing over 75,000

command accounting reports from a 24 hour period.) The display shows two bands of reports: *system commands* (upper band ①)—those executed by either the `root`, `adm`, or various daemon logins and *user commands* (lower band ②)—those executed by ordinary users. The system commands are sorted alphabetically by command name; the user commands are sorted descendingly by the number of times each command was executed. In Figure 2, the tick marks are color- and angle-coded by the user-id. The user-ids are color-coded according to the interactive color scale ③ on the left side of the display. The total number of occurrences for each command is shown on the right side of the display in the form of a bar chart ④, and the total number of occurrences for all commands is shown on the bottom of the display in the form of a stacked histogram ⑤. The slider in the bottom-left corner controls the bin size for the stacked histogram and is currently set at five minutes ⑥.

Many things are apparent from the display in Figure 2. There was a lot of activity for the system commands (top band ①). The commands `chkconfig` and `rpc.mount`, spawned by `sh` (Bourne shell), execute continuously throughout the ten hour period. These involve our network file system (NFS). Another sequence of commands is executed hourly, on the hour. These commands involve accounting and periodic administrative tasks.

The user commands (second band ②) follow a different pattern. The stacked histogram at the bottom of the display shows that there was little user activity before 9, between 10 and 11 and during the noon hour. On this particular day, there was a department seminar between 10 and 11 and a lunch for our visitor. The most popular user command was the `CC` shell script, the front-end for the C++ compiler, which executed 1170 times. Our machine is primarily used for development and therefore the C++ compiler is used frequently.

The first set of user commands were executed by user-id `eick`. He started `ksh` (Korn shell) and read mail at 6:50, executed several other commands at 7:47, and no other commands on this machine during the entire observation interval. Clicking the mouse on the tick mark corresponding to *mailx* command pops up a *browser* window showing the original, source command accounting trace file with the corresponding report centered in it. (The color-

coding of the lines of text in the trace file matches the color-coding of the tick marks.)

There are large bunches of commands executed by user-id `pjl` (Paul Lucas). Those "waves" ⑦ of commands were all started by the `CC` command.[3] The first few were recompiles of selected object files; the compile performed around noon was a complete recompile. By changing the time scale on Figure 2 to show more detail, it is possible to see that many of the commands are executing in parallel, taking advantage of the automatic parallelization on the server.

Some of the commands have tails indicating that they ran for a noticeable amount of time. A few commands have tiny tails, particularly `pjl`'s `make`s and `CC`s, that are at different heights. The height of the tails varies when they would otherwise overlap on the display. A `make` command typically executes several other commands in sequence. On a single processor machine we would expect the commands spawned by the `make` to be executed one after each other with no overlap. The `make` command on our multi-processor compute server can make object files in parallel. The overlapping tails are instances when parallelization occurred, since the commands are executing concurrently. That is, multiple instances of the `CC` and `make` were all started by a single `make` and are executing in parallel on different processors.

4.2 Coding by I/O

The C++ compilations are interesting for performance considerations. Figure 3 shows the display now color- and angle-coded by the I/O character-count, that is, the number of characters read and written by a command. Correspondingly, the color scale ① has changed. Additionally, a *selector* ② restricts the view to display only those commands executed by `pjl` with the tails turned off. The most I/O-intensive commands were `cpp` (the C preprocessor) ③, `cfront` (the C++ compiler) ④, `as1` (the assembler) ⑤, `ln` (the linker) ⑥, and a few other

3. He was compiling *SeeLog* many times.

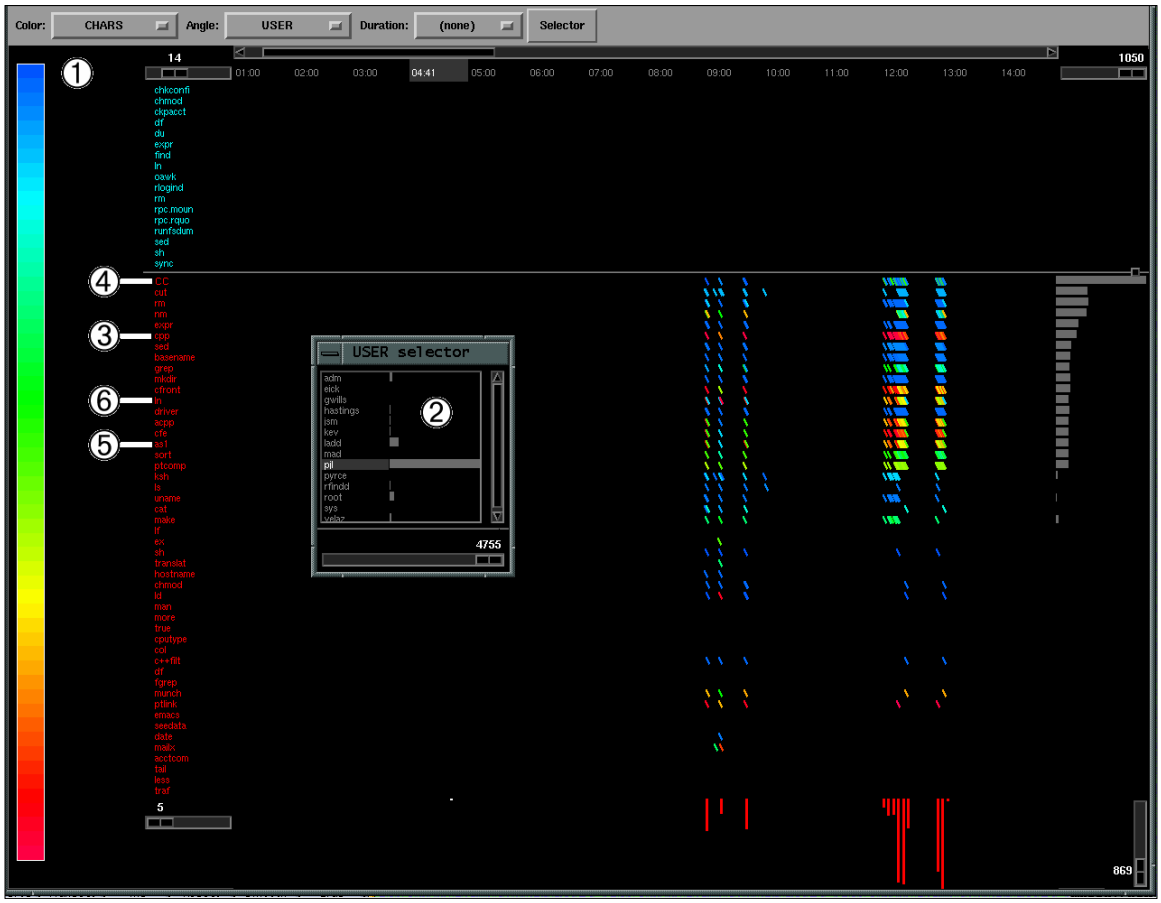

Figure 3. I/O-intensive commands

The tick marks are color- and angle-code to show the amount of I/O character-count of each command. The red tick marks were the most I/O intensive.

commands that executed as part of the compilation process.

4.3 Coding by CPU-time

Commands heavily using CPU time affect machine performance. In Figure 4, the commands are color-coded by CPU-time and angle-coded by user-id with the tails back on. The command using the most CPU time is `traf` ②, a network positioning and visualization system.[11] (It is the last command in the list of user commands.) The command completely occupied one of the four processors on our machine for over ten minutes (614 seconds as shown in the display at the bottom of the color scale ③). Other CPU-intensive commands

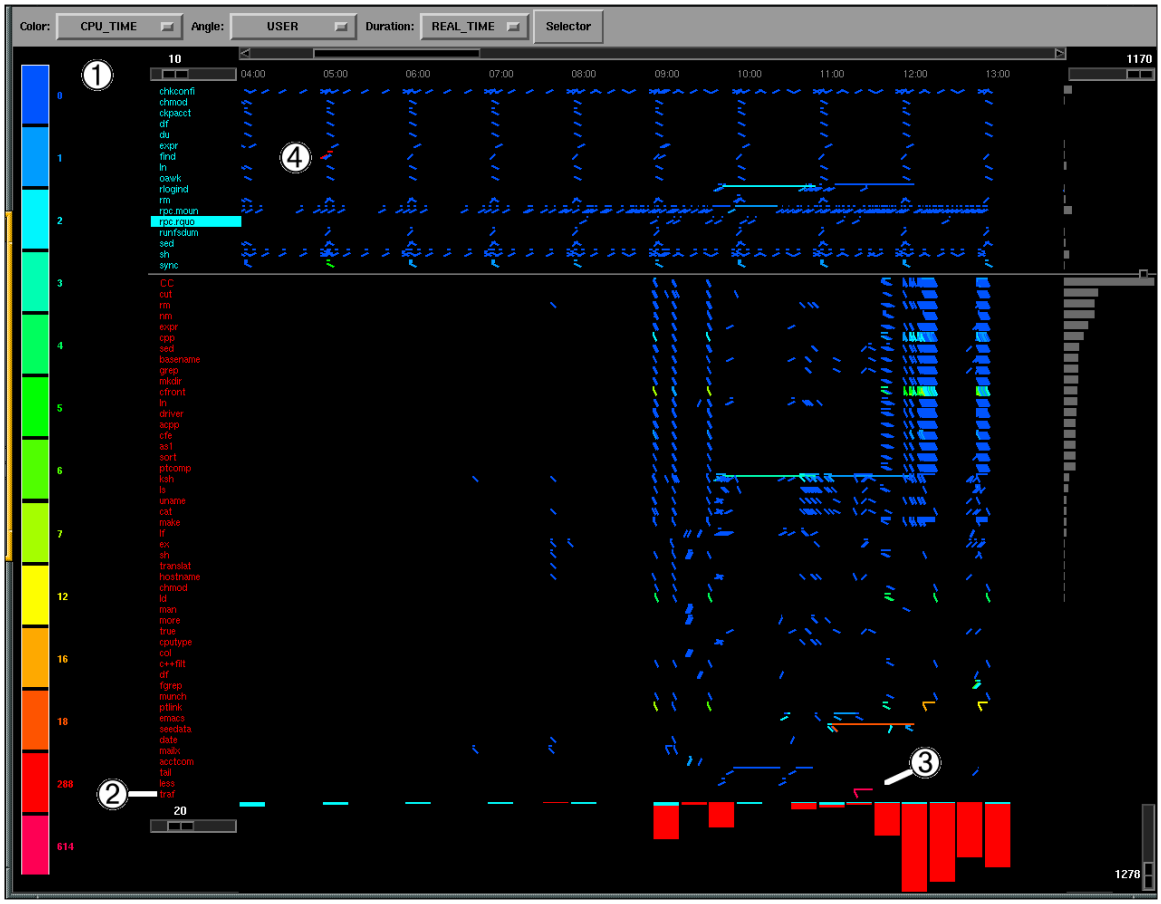

Figure 4. CPU-intensive commands

The tick marks are color- and angle-code to show the CPU-usage of each command. The red (dark) tick marks used the most and the blue (light) the least CPU-time. The biggest CPU-using commands, `traf` and `SeeData`, started at about 11 and are apparent from their red color. The longest running command are ksh's that are indicated by their long tails.

were `ptlink` (part of the C++ compilation process), `cfront`, `ld`, and `find` ④ in the system commands band at 5:00 (nearly 5 CPU-minutes over 7 real-minutes).

4.4 System Security

Commands run with super-user permissions may indicate an unauthorized use of the computer system. Figure 5 shows the display now color-coded by super-user permissions with the commands not run with super-user permissions turned off ②. Some system commands are expected to be run with super-user permissions, so these are of little concern.

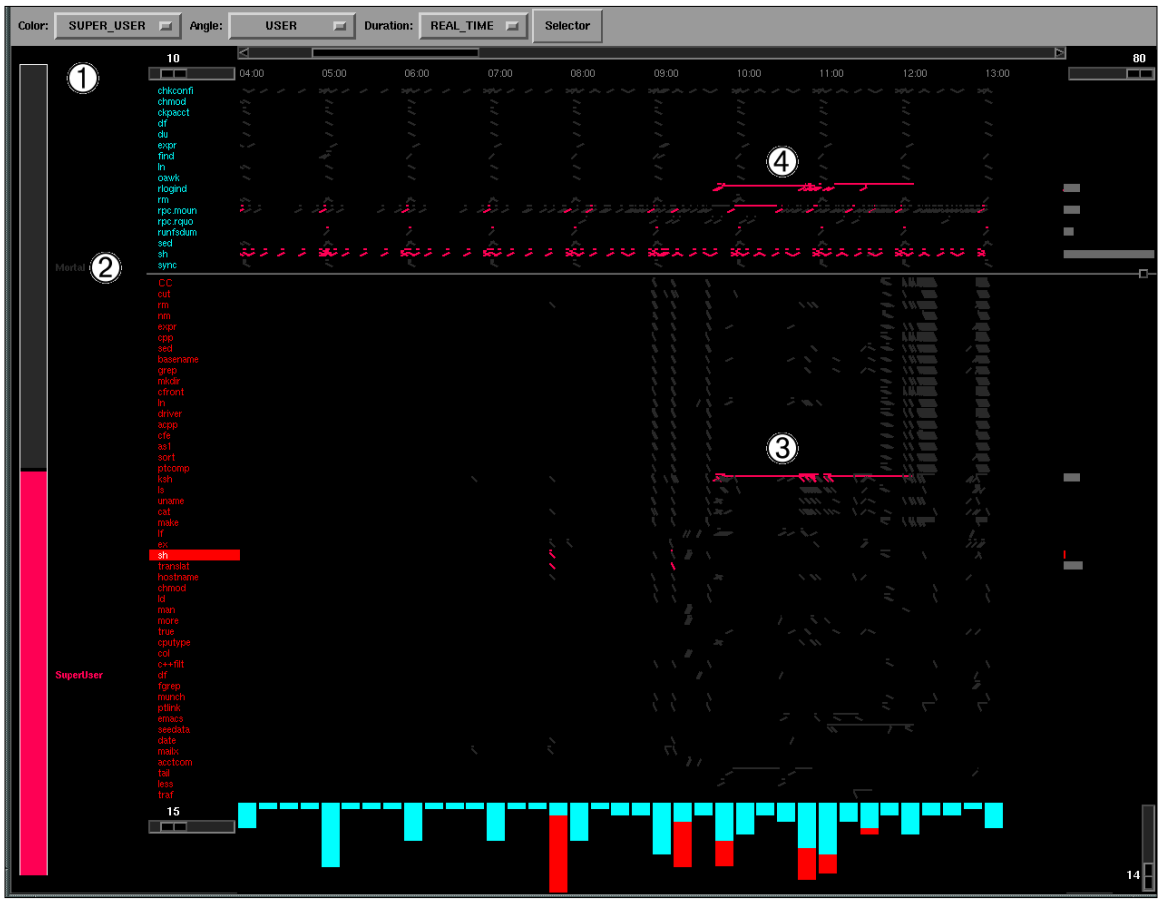

Figure 5. Commands executed with super-user permissions

The commands executed with super-user permissions may be security breaches.

A few user commands, however, were executed with super-user permissions. Starting at 9:47 and continuing until just after noon, several shell commands (`ksh`) were executed ③. These commands are particularly suspicious and caused our system administrator to recheck the integrity of our system. They correlate exactly with remote login daemons (`rlogind`) shown with the system commands ④. After some investigation, he discovered that this behavior is normal and expected. In this case, the rlogin daemon, which runs with super-user permissions, spawned the shells for the user `velaz` to use. Using a *browser* window (not shown) verifies that the shell and login daemon were executed at exactly the same time with the login daemon executing just a bit longer. This is consistent with the expected

behavior on Unix machines.

4.5 Trace File Analysis Results

What have we learned from this analysis?

- System commands execute both continuously and hourly.
- Users started computing seriously at 9pm.
- `CC`, a shell script, is our most frequently executed command.
- The biggest user on the machine, `pjl`, compiled *SeeLog* many times.
- The C++ compilation system is I/O intensive.
- The compute server's parallel `make` facilities work.
- The `traf` program was the most CPU-intensive, completely occupying one of the four processors on our machine for over ten minutes.
- A possible security breach was detected when user commands were executed with super-user permissions.

5. Discussion

Our trace file visualization technique produces a pleasing and informative visual display. It emphasis the time-oriented nature of trace files with positioning and encodes type information using three methods: row, color, and angle. It visually emphases several important characteristics of trace files while showing the reports in context.

5.1 Visual Representation

Visualization is useful for understanding complex trace files provided that they are presented in a clear and easily comprehended display. To achieve this we applied the six key ideas in developing *SeeLog*:

1. Representing each trace file report abstractly as a tick mark.

2. Positioning the tick marks on a grid to display both the chronology and taxonomy of the reports they represent.

3. Coloring and angling the tick marks to encode additional dimensions of type information.

4. Correlating the reports visually, looking for specific patterns including starbursts, waves, alternating sequences, and repetitive vertical patterns.

5. Interacting with selectors to reduce the complexity of the display by filtering the data.

6. Linking the tick marks to the source trace file text through browser windows.

Representing each report as a tick mark neatly solves the spatial separation problem between error reports that occurs in conventional text editors. The positioning of the tick marks on one display provides a complete view, proving an overall impression of the trace file, as well as highlighting any patterns. Coloring and angling facilities pack information onto the display, and the selectors filter the information, preventing it from becoming visually confusing. By linking the tick marks to the trace file through the browser windows, the user is never far from the trace file, yet never becomes lost in it.

5.2 Interaction Techniques

A key aspect of the *SeeLog* system involves the interactive filtering and focusing techniques. For complicated trace files *SeeLog* displays may become overly cluttered. By manipulating the controls, the analyst filters the unwanted information from the display and thereby eliminates clutter and reveals patterns. There are six interactive controls on the display:

1. The time scale sliders set the width of the display and select the time-period to be viewed. Using these controls the analysts may start by looking at the whole trace and zoom in on an interesting region.

2. The color scale is interactive and users may deactivate colors using the mouse and thereby reduce the display clutter. (See Figure 5.)

3. Interactive selectors also reduce visual clutter and focus the display, thereby revealing patterns. (See Figure 3.)

4. The bin sizes for the totals by time period are controlled by a slider. As the analysts manipulates the slider *SeeLog* continuously reaggregates. This facilitates looking for cyclic patterns at many different time scales. (See Figure 4.)

5. The barplots showing the total number of reports by type are is also interactive. Often the total for one command dominates the scale so that the other bars are barely visible. By adjusting the slider the analyst may distort the bar sizes making the smaller bars more visible, thereby revealing secondary patterns.

6. The display itself is interactive; clicking on any tick mark causes a browser window to appear showing the text in the trace file corresponding to the tick mark. (See Figure 2.) This drill-down capability is particularly useful for relating overall patterns to individual reports in the trace file.

5.3 Comparison With Other Techniques

How were trace files analyzed before we developed *SeeLog*? Most were not! Only and when there was a performance problem would anyone attempt the difficult task of looking at the files. We are not aware of any other tools that enable one to look at a large trace file and extract information with reasonable effort. *SeeLog*, on the other hand, provides an overview of a trace file and permits rapid analysis.

6. SeeLog Status

SeeLog, as of this writing, is undergoing test use in the 5ESS development and testing organization of AT&T. Its primary use is as an aid for debugging the large trace files generated during switch stability runs. Analysis that once took many hours is consistently being completed in half an hour or less. A by-product of the increased speed is that software engineers now have time to investigate many more additional problems in each trace file and not just the most critical ones. In addition to the set of problems that conventional

techniques would have led the engineer to find eventually, our technique enables the discovery of additional problems and correlations that conventional techniques would not.

7. Summary

We have described a graphical technique for visualizing large, computer-generated trace files of time-stamp typed messages and applied it to visualize a trace file of Unix commands executed on an multi-processor computer server. Our technique is embodied in an interactive system that displays trace file reports on a type-by-time grid, thereby preserving the temporal aspects of the reports. All of the reports are displayed on a single grid as tick marks, using position, color, and angle to encode the type, time, attributes and subattributes of each report. We have developed a suite of interactive techniques for manipulating the display, filtering display clutter, moving through time, aggregating, and probing individual reports for detail to aid in trace file analysis. We have applied our technique for successfully analyzing special purpose trace files with tens of thousands of reports and illustrate its usage by analyzing Unix command accounting trace files.

8. Acknowledgments

Craig Votava suggested the application of *SeeLog* to Unix command accounting trace files. An early version of this paper appeared in.[12]

REFERENCES

1. Stephen G. Eick, Joseph L. Steffen, and Eric E. Sumner, Jr. "Seesoft™—A Tool for Visualizing Line-Oriented Software Statistics." *IEEE Transactions on Software Engineering*, 18(11), November 1992, pp. 957–968.

2. Stephen G. Eick, Michael C. Nelson, and Jeffrey D. Schmidt. "Graphical Analysis of Computer Log Files." *Communications of the ACM*, 37(12), December 1994, pp. 50-56.

3. Robert W. Scheifler and James Gettys. "The X Window System." *ACM Transactions on Graphics*, 5(2), August 1983. pp. 57–69.

5. Stephen Jose Hanson, Robert E. Kraut, and James M. Farber, "Interface Design and Multivariate Analysis of UNIX Command Use," *ACM Transactions on Office Information Systems*, 2(1), March 1984, pp. 42-57.

5. Robert E. Kraut, Stephen Jose Hanson, and James M. Farber, "Command Use and Interface Design," *ACM CHI '83 Conference Proceedings*, pp. 120-124, December, 1983.

6. M. T. Heath and J. A. Etheridge, "Visualizing the Performance of Parallel Programs," *IEEE Software*, September, 1991, pp. 29–39.

7. D. A. Reed, R. A. Aydt, R. J. Noe, P. C. Roth, K. A. Shields, B. W. Schwartz, and L. F. Tavera, "Scalable Performance Analysis: The Pablo Performance Analysis Environment," *Proceedings of the Scalable Parallel Libraries Conference*, A. Skjellum. Editor, IEEE Computer Science Press, 1993, pp. 104-113.

8. Wendy Waugy. *5XPRESS Switch Maintenance Expert Users Guide*. Personal communications, November 1993.

9. Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger. *The AWK Programming Language*. Addison-Wesley, 1988. ISBN 0-201-07981-X

10. Lawrence Wall and Randal L. Schwartz. *Programming perl*. O'Reilly & Associates, Inc., 1990. ISBN 0-937175-64-1

11. Stephen G. Eick and Graham J. Wills. "Navigating Large Networks with Hierarchies," *Visualization '93 Conference Proceedings*, San Jose, CA, October 25, 1993. pp. 204–210.

12. Stephen G. Eick and Paul J. Lucas, "Graphically Analyzing Computer Log Files," *Interface '94 Conference Proceedings*, Raleigh, North Carolina, June, 1994, pp. 143-145.