Visually Monitoring Program Execution

(Extended Abstract)

Steven P. Reiss
Department of Computer Science
Brown University
Providence, RI 02912
spr@cs.brown.edu (401) 863-7641
June 4, 1991

ABSTRACT

In this paper we describe a suite of tools we have developed as part of the FIELD programming environment. These tools provide dynamic views of executing programs. The first provides a view of dynamically allocated storage. The second shows the files being accessed by the application. The third provides a variety of performance measures on an executing program. These tools operate by replacing the standard system library. With shared libraries on Sun OS, this allows the tools to be used on existing programs.

1.0 Motivation

Both program understanding and program debugging require providing the user with detailed information about the static and dynamic behavior of the underlying application. The FIELD environment has offered a variety of tools for this purpose [REISS90a, REISS90b, REISS90c]. These include a variety of tools that offer static views of the program and some tools that provide a limited view of the dynamics of execution. While the static tools have demonstrated their utility over a wide range of systems, the dynamic tools have been much more limited.

The approach used in the previous dynamic tools in FIELD was to monitor the application through the system debugger, either *dbx* or *gdb*. This approach introduces considerable overhead, but it has the advantage that it is totally non-intrusive: it doesn't require any effort on the part of the programmer and it works on existing programs. In developing a new set of dynamic visualization tools for FIELD, we wanted to achieve much better performance and at the same time minimize the intrusiveness of monitoring. We also wanted to fine-grained monitoring.

Our initial set of dynamic monitoring tools includes three visualizers. The first provides a dynamic view of the heap. It shows how the application is using memory by providing a map of the heap. It uses colors to either show the temporal nature of the heap or the sizes of allocated blocks. The second visualizer provides insight into the application's use of files. It shows all file

input/output operations over the life of the application. Colors can be used to show temporal access patterns or to show the sizes of I/O operations. The third visualizer offers a variety of performance data about a running application. This data, gathered primarily from the getrusage system call by the application, is split into a selection of statistics that the user can display and that will dynamically update as the application runs.

This work builds on the significant body of existing tools for dynamic program visualization. FIELD already offers tools to dynamically show data structures and to visualize execution in various static program displays (editors, call graphs, class hierarchies) [REISS89]. BALSA [BROWN85]and TANGO [STASKO89] provide algorithm animations. These are high-quality, specialized visualizations of programs that are driven by events that are explicitly inserted into the application by the programmer. The AE system [LARUS90] uses a modified compiler and an intelligent run time library to detect and generate assembler-level traces of program execution. PIXIE from MIPS [MIPS88] works by modifying the binary file to insert additional instructions to monitor the number of times each instruction is executed. We have taken a similar idea in the *iprof* profiler we developed for the Sparc architecture. Here however, we modify the binary at run time, and offer the capability to use a shared memory segment for the count buffers so that a dynamic profile can be provided. Many other tools have been developed for program visualization. These range from simple tools that mapped memory to the display, to tools that monitor resource usage as reported by the operating system, to tools that monitor distributed processes using hooks in the synchronization mechanisms, to tools that generate operating system level traces, to the wide variety of tools that are provided by interpreters.

Our efforts differ from these efforts in two ways. First of all, our method of monitoring is less intrusive and yet efficient. It is suitable for large systems and can report application private information. For example, we have used the new visualizers on FIELD itself. Secondly, we have put significant effort into providing visualizations that are dynamic, that convey detailed information, and that are flexible enough to handle a wide range of user requests.

2.0 Monitoring

In order to drive dynamic program visualizations, information has to be generated by the application and received by the visualizer tools. We call this monitoring. We had two basic goals in developing a monitoring strategy for visualization. The first is that the monitoring be done with as little intrusion into the user code and the application as possible. The second is that the monitoring not degrade program performance substantially.

We require that the application and the visualizer run in separate address spaces. This allows the application to be run with or without monitoring and isolates the visualization tool from the application. However, it means that information from the monitoring has to be communicated between the two processes. We decided to use the broadcast message facility provided by FIELD as the basis for this communication. This uses TCP/IP stream sockets and supports both synchronous and asynchronous messages. It also provides automatic buffering of messages allowing the sender to get ahead of the receiver where necessary, useful when the visualization graphics are too slow.

We first attempted to use the FIELD message facility directly for monitoring. This demonstrated that messaging was fast enough to accommodate most applications and most monitoring. However, it tended to distort the monitoring results because the FIELD message client code is relatively sophisticated and can make significant use of UNIX memory allocation and files, two of the items we were monitoring.

To avoid these problems, we developed a simple socket interface that resides in the application and a separate FIELD service (*monserver*) that communicates between the application being monitored and the FIELD message server. This communications server offers three benefits. First of all, it greatly simplifies the code that has to reside in the application being monitored. Secondly, it provides a buffer, allowing the application to run as fast as possible. Finally, it allows the monitoring to be controlled by FIELD tools.

We generate the monitoring messages by providing our own library that replaces the system and standard library calls with ones that do monitoring. This requires that the application be built to include this library if monitoring is to be done. To monitor memory allocation, our library replaces the library version of malloc, free, and realloc. To monitor I/O, our library replaces all the I/O system calls with versions that use syscall. In both cases, before a message is sent, the execution and real time used by the application are checked and additional messages are sent out if these have changed from their previous values. Performance monitoring is done as an extension of this timing mechanism. At each potential message call, the resources used by the application are determined using getrusage (2), and, if these have changed since the previous performance message, a resource message is sent out. Monitoring is controlled in this library via the MONITOR environment variable.

The shared library facility provided by Sun allows us to simplify monitoring. The monitoring library can be spliced in between the application and the standard library. This allowed us to implement a simple command, *hmon*, that takes a set of options and a command line. It uses the options to set the MONITOR environment variable, and executes the specified command with the monitoring library.

3.0 Heap Visualization

We had several objectives in mind when we constructed our heap visualization tool: to provide a visual representation of an application's memory usage, to provide information as to when and how memory was used by the application, and to provide tools to allow the programmer to identify memory allocation problems such as bottlenecks and memory leaks.

To achieve these objectives, the heap visualizer provides the user with a two-dimensional map of heap memory. An example is seen in figure 1. The different patterns correspond to different colors. The visualizer allows the colors to be used to represent either allocation sizes or allocation times. Allocation time can represent either real time spent by the application, execution time of the application, virtual time, or user time. Virtual time is a count of the number of memory allocations or frees. User time is represented by a counter that is incremented when the user identifies a time mark. This is done by clicking on the Mark button in the Time menu.

Two strings are currently associated with each block, the source of the allocation and the type allocated. By default, the type is empty and the source is the address in the application of the call to malloc. This data can be provided by user variants or extensions to malloc via simple call to the monitoring routines.The user can click on any block of storage to request this information.

In addition to the main visual heap display, this tool offers two auxiliary windows. The first displays the current palette, indicating the current coding of colors or fill patterns on the display. The second is a statistics window. It displays current statistics on memory and malloc usage and is continually updated as the program being monitored runs.

4.0 I/O Visualization

Our objectives in constructing the input-output visualizer was to provide a facility that would provide the programmer with insight into how his program interacted with the outside world by showing file operations. We wanted the programmer to be able to locate and understand I/O bottlenecks. We also wanted to detect simple file-related problems such as failure to close a file or opening files more times than necessary.

To accomplish these ends, we provide a scrollable display showing all the files that are or were successfully opened, including those that were open when the process started running. Each file has the associated name, if it is known, or the type of file, i.e. SOCKET, PIPE or TTY, if the name is not known. It also includes the current file length and a visual map of operations on the file. The map consists of a scrollable horizontal bar representing the file, with each operation represented as a colored region in the bar. The location of the region corresponds to the location of the read or write in the file. An example is seen in figure 2.

Colors and fill patterns can be used to indicate either the size of the I/O operation or the time at which the operation was performed, as well as whether the operation was a read or a write. The user can click with the mouse on any portion of the file display to get detailed information. A variety of different display options allow programmers to customize the display to their needs. As with the heap viewer, the input-output viewer provides a palette and a statistics window.

5.0 Performance Visualization

In addition to providing information about the heap and files, we wanted to provide the programmer with performance data. The initial set of statistics that we support are those provided by the getrusage system call. These include page faults, swaps, an approximation to memory usage, block input and output operation counts, signals, and context switches. The system is designed so that new statistics are easy to add. The statistics are displayed graphically, with the X axis representing time and where any number of statistics can be displayed simultaneously. An example display can be seen in figure 3.

6.0 Experiences and Future Work

The various viewers have demonstrated their utility as part of the FIELD environment. The heap viewer has been the most useful thus far. It has been used for finding memory leaks, for understanding memory usage behavior, and for understanding the behavior of different memory allocation strategies. The I/O viewer has seen less usage. Still, it has proven valuable in detecting multiple opens of files and of the efficiency of interprocess communication based on sockets. The performance tool has proven less useful. This is primarily because the most relevant numbers, those dealing with paging behavior and usage, are not accurate when reported by getrusage and thus the resultant displays are not necessarily meaningful.

The monitoring library, monitor server, and the visualization tools were all implemented in C in conjunction with the FIELD tools. The monitoring library involved about 1500 lines of new code. The monitor server is about 600 lines of code. Each of the visualization tools is between 4000 and 5000 lines of code.

Our future efforts will be to extend the monitoring facilities to provide additional services. One extension we currently provide on Sparcstations is to offer the *iprof* instruction count profiling capabilities from within the monitoring library. The *imon* command, similar to *hmon,* takes an existing program and executes it with the monitoring library options set that an instruction count profile is generated. We are currently working on monitoring function calls and returns and are trying to determine feasible means for effectively monitoring changes to complex data structures.

7.0 References

[Brown85] Marc H. Brown and Robert Sedgewick, “Techniques for algorithm animation,” *IEEE Software* Vol. 2(1) pp 28-39 (1985).

[Larus90] James R. Larus, “Abstract Execution: A technique for efficiently tracing programs,” U. Wisconsin-Madison Computer Science Dept. TR 912 (February 1990).

[MIPS88] MIPS Computer Systems, Inc,. *RISCompiler Languages Programmer’s Guide*. December 1988.

[Reiss90a] Steven P. Reiss, “Interaction with the FIELD environment,” *Software Practice and Experience* Vol. 20(S1) pp. 89-115 (June 1990).

[Reiss90b] Steven P. Reiss, “Connecting tools using message passing in the FIELD environment,” *IEEE Software* Vol. 7(4) pp. 57-67 (July 1990).

[Reiss90c] Steven P. Reiss and Scott Meyers, “FIELD support for C++”, *Proc. USENIX C++ Conference*, pp. 293-300 (April 1990).

[Stasko89] John T. Stasko, “TANGO: A framework and system for algorithm animation,” *IEEE Computer* Vol. 23(9) pp 27-39 (September 1990).

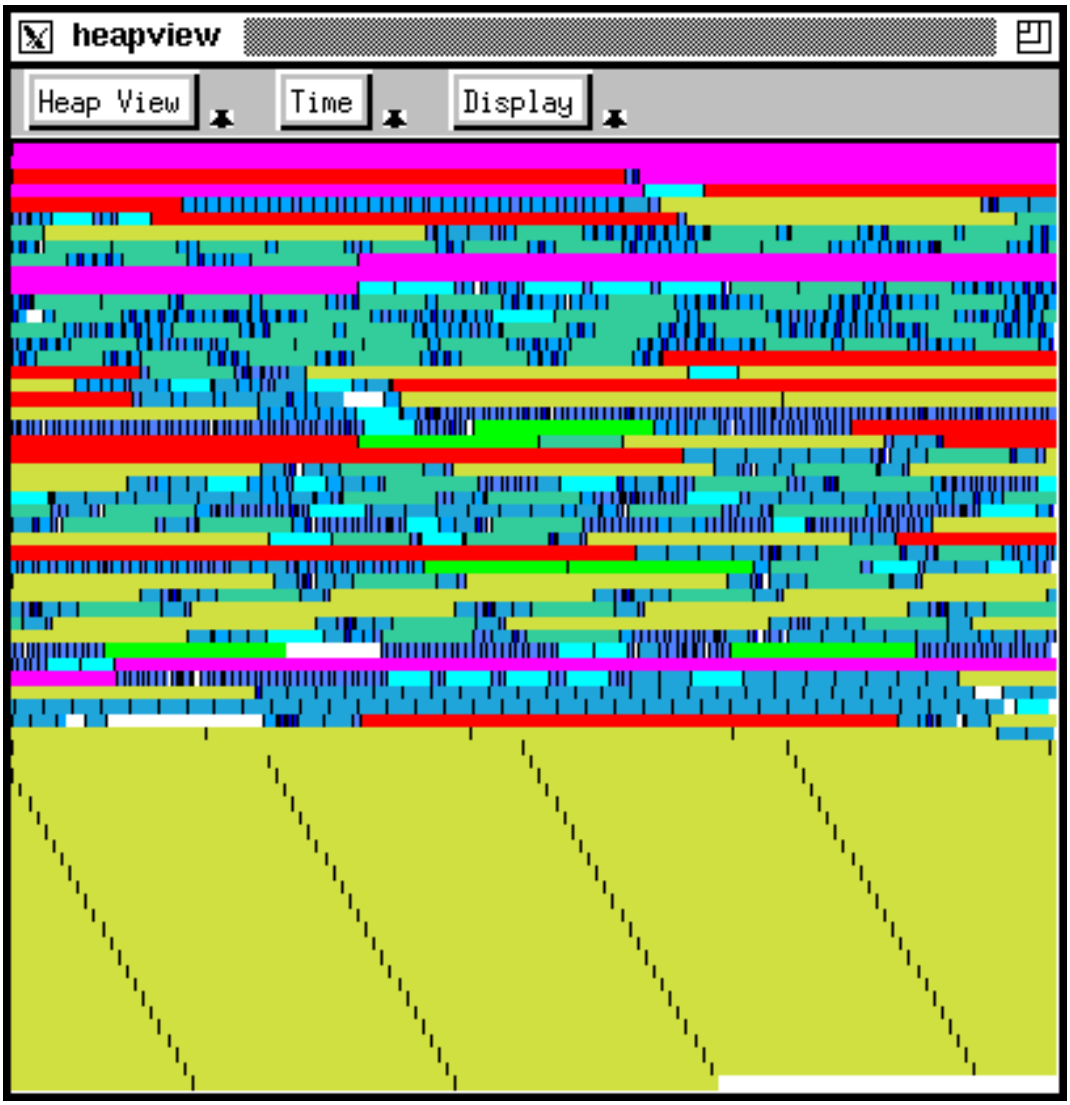

Figure 1: Example of Heap Visualizer

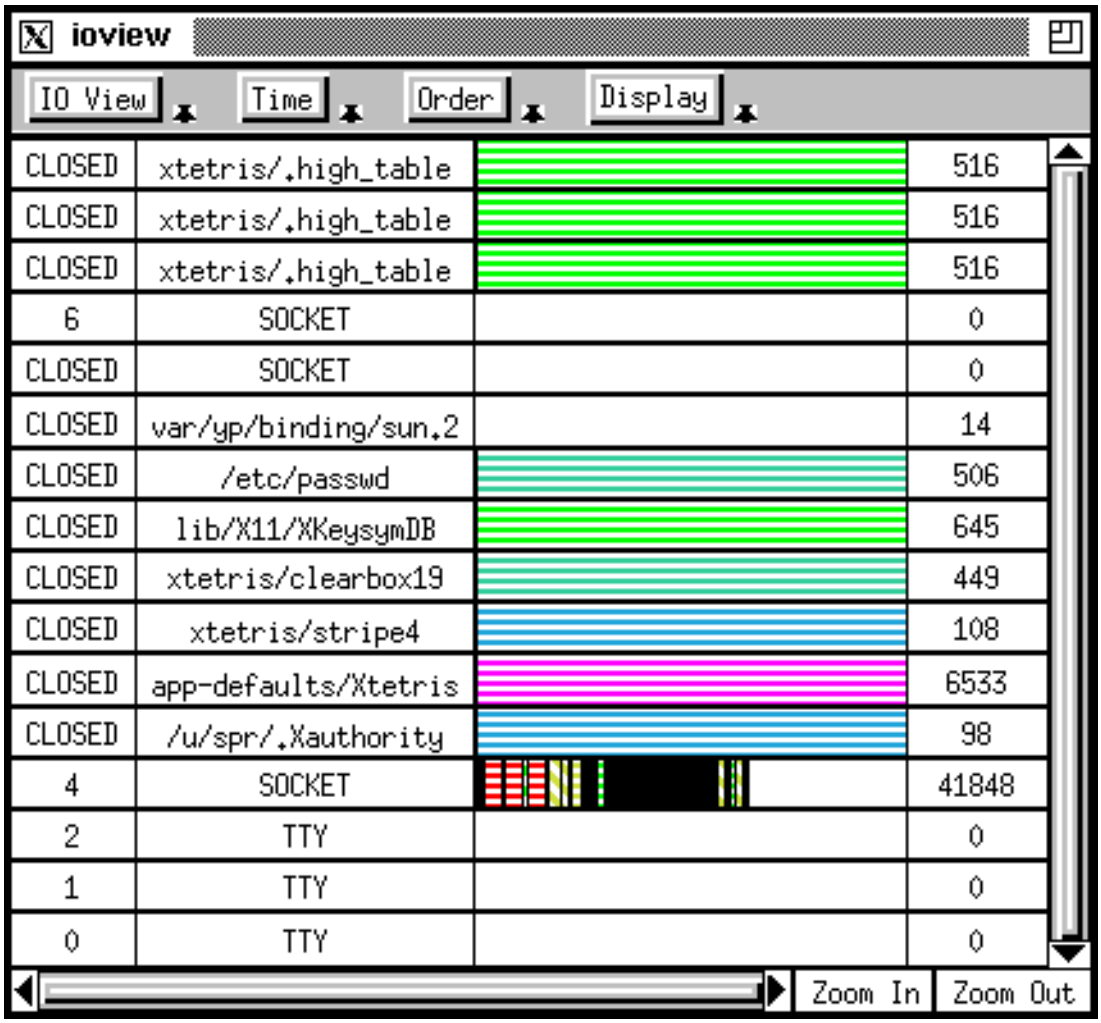

Figure 2: Example of Input/Output Visualizer

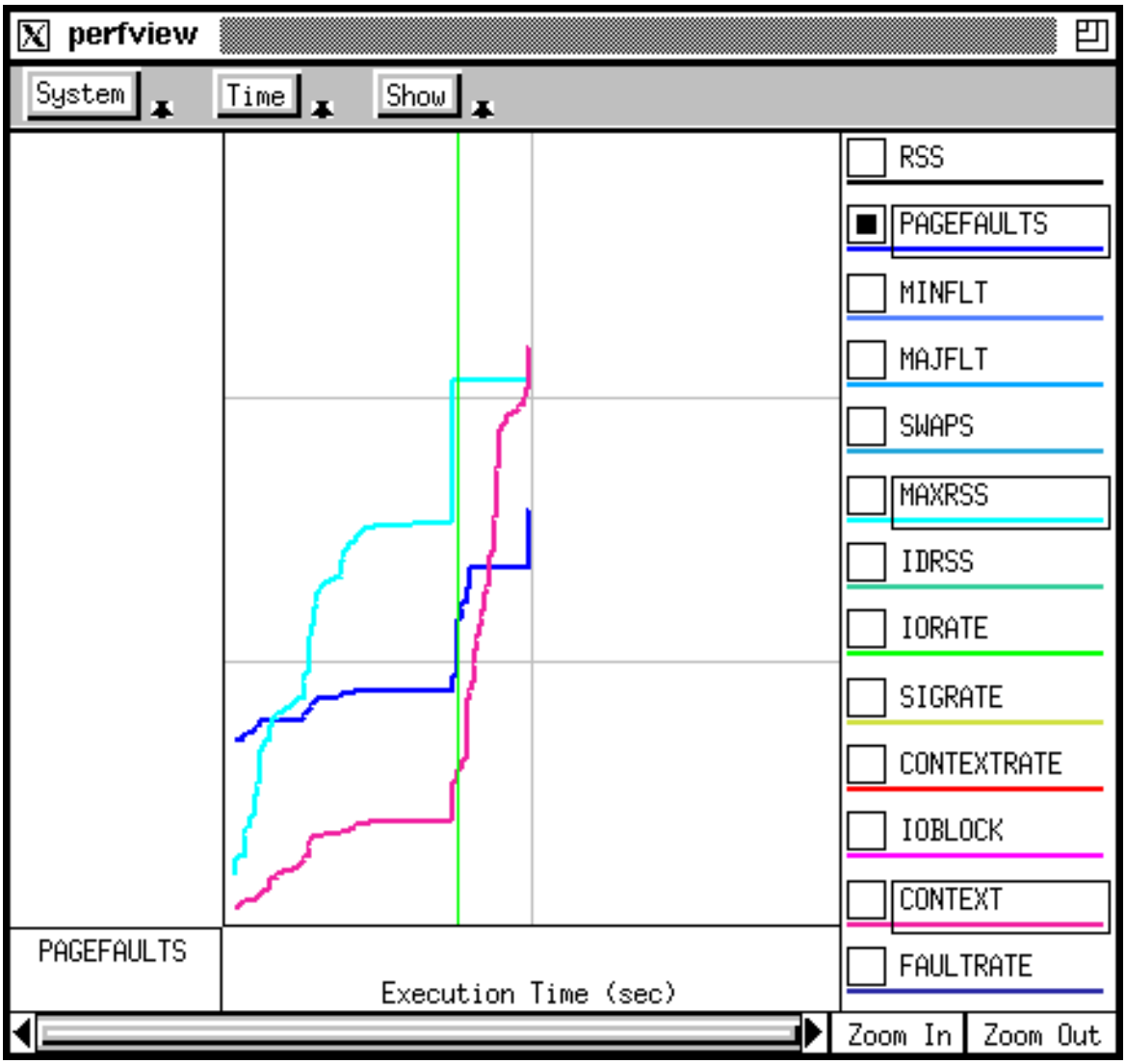

Figure 3: Example of Performance Visualizer