Strata-Various:* Multi-Layer Visualization of Dynamics in Software System Behavior

Doug Kimelman, Bryan Rosenburg, Tova Roth

IBM Thomas J. Watson Research Center
Yorktown Heights, NY 10598

Abstract

Current software visualization tools are inadequate for understanding, debugging, and tuning realistically complex applications. These tools often present only static structure, or they present dynamics from only a few of the many layers of a program and its underlying system. This paper introduces "PV", a prototype program visualization system which provides concurrent visual presentation of behavior from all layers, including: the program itself, user-level libraries, the operating system, and the hardware, as this behavior unfolds over time. PV juxtaposes views from different layers in order to facilitate visual correlation, and allows these views to be navigated in a coordinated fashion. This results in an extremely powerful mechanism for exploring application behavior. Experience is presented from actual use of PV in production settings with programmers facing real deadlines and serious performance problems.

1 Visualization of Software Behavior

To truly understand any realistically complex piece of software, for purposes of debugging or tuning, one must consider its execution-time behavior, not just its static structure. *Actual* behavior is often far different from expectations, and often results in poor performance and incorrect results. Further, the ultimate correctness and performance of an application (or lack thereof) arises not only from the behavior of the program itself, but also from activity carried out on its behalf by underlying system layers. These layers include user-level libraries, the operating system, and the hardware. Finally, problems often become apparent only when one considers the interleaving of various kinds of activity, rather than cumulative activity summaries at the end of a run. Thus, for debugging and tuning applications in a realistically complex environment, one must consider behavior at numerous layers of a system concurrently, as this behavior unfolds over time.

Clearly, any textual presentation of this amount of information would be overwhelming. A visual presentation of the information is far more likely to be meaningful. Information is assimilated far more rapidly when it is presented in a visual fashion, and trends and anomalies are recognized much more readily. Further, animations, and views which incorporate time as an explicit dimension, reveal the interplay among components over time.

With an appropriate visual presentation of information concerning software behavior over time, one can first survey a program execution broadly using a large-scale (high-level, coarse-resolution) view, then narrow the focus as regions of interest are identified, and descend into finer-grained (more detailed) views, until a point is identified for which full detail should be considered.

* :-)

Further, displays which juxtapose views from different system layers in order to facilitate visual correlation, and which allow these views to be navigated in a coordinated fashion, constitute an extremely powerful mechanism for exploring application behavior.

2 PV — A Program Visualization System

PV, a prototype program visualization system developed at IBM Research, embodies all of the visualization capabilities proposed above. Success with PV in production settings and complex large-scale environments has verified that these capabilities are indeed highly effective for understanding application behavior for purposes of debugging and tuning.

Users often turn to program visualization when performance is disappointing — either performance does not match predictions, or it deteriorates as changes are introduced into the system, or it does not scale up (and perhaps even worsens!) as processors are added in a multiprocessor system, or it is simply insufficient for the intended application.

With PV, users watch for trends, anomalies, and interesting correlations, in order to track down pressing problems. Behavioral phenomena which one might never have suspected, or thought to pursue, are often dramatically revealed. A user continually replays the execution history, and rearranges the display to discard unnecessary information or to incorporate more of the relevant information. In this way, users examine and analyze execution at successively greater levels of detail, to isolate flaws in an application. Resolution of the problems thus discovered often leads to significant improvements in the performance of an application.

PV shows hardware-level performance information (such as instruction execution rates, cache utilization, processor element utilization, delays due to branches and interlocks) if it is available, operating-system-level activity (such as context switches, address-space activity, system calls and interrupts, kernel performance statistics), communication-library-level activity (such as message-passing, inter-processor communication), language-runtime activity (such as parallel-loop scheduling, dynamic memory allocation), and application-level-activity (such as algorithm phase transitions, execution time profiles, data structure accesses).

PV has been targeted to shared-memory parallel machines (the RP3[7]), distributed memory machines (transputer clusters running Express), workstation clusters (RISC System/6000 workstations running Express), and superscalar uniprocessor workstations (RISC System/6000 with AIX).

PV is structured as an extensible system, with a framework and a number of plug-in components which perform analysis and display of event data generated by a running system. It includes a base set of components, and users are encouraged to add their own and configure them into networks with existing components. Novice users simply call up pre-established configurations of components in order to use established views of program behavior.

Figures 1 through 6 show some of the many views provided by PV. Section 4 describes some of these views in detail and discusses their use.

3 AIX Trace

PV is trace-driven. It produces its displays by continually updating views of program behavior as it reads through a trace containing an execution history. A trace consists of a time-ordered sequence of event records, each describing an individual occurrence of some event of interest in the execution of the program. Typically, an event record consists of an event type identifier, a timestamp, and some event-specific data. Events of interest might include: sampling of a cache miss counter, a page fault, scheduling of a process, allocation of a memory region, receipt of a message, or completion of

some step of an algorithm. A trace can be delivered to the visualization system live (possibly over a network), as the event records are being generated, or it can be saved in a file for later analysis.

The standard AIX system (IBM's version of UNIX), as distributed for RS/6000s, includes an embedded trace facility. AIX Trace[6], a service provided by the operating system kernel, accepts event records generated at any level within the system and collects them into a central buffer. As the event buffer becomes full, blocks of event records are dumped to a trace file, or dumped through a pipe to a process, e.g. for transmission over a network. Alternatively, the system can be configured to simply maintain a large circular buffer of event records that must be explicitly emptied by a user process.

Comprehensive instrumentation within AIX itself provides information about activity within the kernel, and a system call is provided by which user processes can provide event records concerning activity within libraries or the application. On machines incorporating hardware performance monitors, a device driver can unload hardware performance data, periodically or at specific points during the execution of an application, and generate AIX event records containing the data.

The variant of PV that is targeted to AIX workstations is based on AIX Trace. Unless otherwise noted, all of the applications discussed in this paper were run on AIX RS/6000 workstations with AIX Trace enabled. Traces were taken during a run of the application and saved in files for later analysis.

For the applications discussed here, tracing overhead was negligible — less than 5% in most cases. In no case was perturbation great enough to alter the behavior being investigated. Trace file sizes in all cases were less than 16 megabytes.

4 Views in Action — Experience with PV

This section describes some of the many views provided by PV, and explains their use, by way of examples of actual experience with PV.

PV has been applied to a number of different types of application across a number of domains, including: interactive graphics applications written in C++, systems programs such as compilers written in C, computation-intensive scientific applications written in Fortran, I/O-intensive applications written in C, and a large, complex, heavily-layered, distributed application written in Ada.

4.1 Views of Process Scheduling and System Activity

In one example, the developers of "G", an interactive graphics application,[1] were concerned that it was taking 12 seconds from the time that the user entered the command to start the application, until the time that the main application window would respond to user input. They suspected that a lot of time was being lost in the Motif libraries.

End-of-run summaries showed that 51 seconds out of a 97 second run were spent idle, but these summaries provided no indication of how many of these idle seconds were in fact warranted, perhaps waiting for user input, and how many were somehow on the critical path for the application. Profiles of time spent in various functions, and perusal of thousands of lines of detail in textual reports, would not have been helpful.

PV views showing process scheduling alongside operating system activity immediately highlighted the nature of this performance problem. The scheduling view consists of a strip of color growing to the right over time, with color used to indicate which process was running at any instant

[1] "The stories you are about to hear are true. Only the names have been changed to protect the innocent."

in time. (Figure 2 shows this view in a window titled "AixProcess | ColorStrip".) The activity view consists of a similar strip of color, in which color is used to indicate what activity was taking place at any instant in time. (Figure 2 shows this view in a window titled "AixSystemState | ColorStrip".) The two views can show the same time spans, and they can be aligned so that a point in one view corresponds to the same instant in time as the point immediately above or below it in the other view. Further, the two views aligned in this way can be navigated in a coordinated fashion — when the user zooms in on either view, expanding a region of interest in order to reveal greater detail, the other view expands the same region of time automatically.[2]

(PV provides a number of other views which can be aligned and navigated in the same fashion, including views showing kernel performance statistics, hardware performance statistics, which loop of a function is currently active, or which user-defined phase of an algorithm is currently being executed.)

By viewing behavior as it unfolded over time, it was apparent that the two processes of application "G" (shown in Figure 2 as light pink and salmon color) were not even running for much of the 12 seconds that they should have been rushing to establish the main application window (!). Further, it wasn't even the X server process (light green) that was running instead of them. In fact the system was idle (dark purple) much of the time. Thus, 5 of the 51 idle seconds noted above were occurring during startup and hence were on the critical path. Finding the point on the scheduling view where a process of application "G" went idle, zooming in to show greater detail, and dropping down to the activity view, revealed the cause of the idle time: system calls to examine a number of files were causing large delays. Having narrowed the focus to a very small window in time using the graphic views, a view was opened showing the detailed textual trace report. As each event is displayed graphically in other views, this view highlights the corresponding line in the report. With this level of detail it was immediately obvious that startup information had inadvertently been scattered across a number of files which might well be remote-mounted and thereby incur significant access penalties.

Without the visual correlation facilitated by juxtaposition of views and coordinated navigation, it would have been much harder to make the connection between the various aspects of this performance problem. At the very least, it would have taken much longer by any less direct means.

4.2 Views of Memory Activity and Application Progress

In another example, PV views revealed a number of memory-related problems in "A", a compiler. Each view in this case is rectangular, with each position along the horizontal axis corresponding to some region of a linear address space (the size of the region depends on scale of the display). In one view, color is used to represent the size of a block of memory on the user heap. (Figure 5 shows this view in the upper window titled "AixMalloc | OneSpace".) In another view, color is used to show the source file name or line number that allocated the block. (Figure 5 shows this view in the lower window titled "AixMalloc | OneSpace".) In a third view, color is used to represent the state of a page of the data segment of the user address space. (Figure 5 shows this view in the window titled "AixDataSeg | OneSpace".) For purposes of correlation, the views are configured to show the same range of addresses, and they are aligned so that a given address occurs at the same horizontal position in each view. As well, zooming in on a region in one view automatically causes the corresponding zoom operation in the other view.

Each of these views in Figure 5 is split into an upper half and a lower half, each representing part of the data segment of the address space of compiler "A". The left edge of the upper half

[2]Complete detail concerning the contents of these views can be found in a lengthy technical report [8].

represents address 0x24200000; successive points to the right along this half represent successively higher addresses; and the right edge represents address 0x24600000. Thus, the upper half of these views represent 2MB of the data segment. Similarly, the lower half of these views represents an expanded view of the 248KB from 0x244CAE35 to 0x2448E571. The black guidelines show where the region represented by the lower half of a view fits into the region represented by the upper half.

These views showed a number of wastes of memory, none of which could technically be classed a "leak". Rather, they were "balloons" — still referred to, but largely full of empty space. In one case, the heap views showed that every second page of the heap was not being made available to the end user (shown in Figure 5 "AixMalloc | OneSpace" as alternating green and white blocks in the lower half of the view), yet the corresponding positions on the data segment view showed clearly that *every* page was being faulted in (shown in the lower half of Figure 5 "AixDataSeg | OneSpace" as all magenta). The heap views also showed that all of the blocks in question were of the same size. Having identified blocks of a particular size as being problematic, the source code for the allocator was quickly inspected, with particular attention to the treatment of blocks of the problematic size. It rapidly became apparent that, in certain situations, half of the heap was being left empty due to an unfortunate interaction between user code, the heap memory allocator, and the virtual memory system.

In another case, a static array was declared to be enormous. This was felt to be acceptable because real memory pages were never faulted in unless they were required for the size of the program being compiled. However, the data segment view emphasized that the array *did* occupy address space, and this became noteworthy when the compiler could not be loaded on smaller machine configurations, even though only moderate-sized programs needed to be compiled.

Finally, late in the run of this compiler, pages began flashing in and out of the data segment view. Glancing at the system activity view (described earlier), during the time that the page flashing was occurring, allowed this behavior to be correlated to periods of excessive disclaiming and subsequent reclaiming of pages by the compiler.

An application phase view, which provides a roadmap to the progress of an application, allowed this thrashing in the address space to be attributed directly to the offending phase of the compiler. The application phase view consists of a number of strips of color, as in the process scheduling and system activity views described earlier. (Figure 5 shows this view in the window titled "AixPhase | ColorStrip".) The strips are stacked one on top of the other, and they grow to the right together over time. The color of the top strip shows which user-defined phase of the application is in progress at any instant in time. The color of successively lower strips shows successively deeper sub-phases nested within the phases shown at the corresponding positions on the higher strips. (This view can be driven by instrumentation in the form of simple event generation statements inserted manually or automatically into the source, or by procedure entry and exit events generated using object code insertion techniques.)

In the case of compiler "A", correlation in time between the application phase view and the data segment view immediately made it clear that a back-end code generation phase (shown in Figure 5 as light green) was responsible for the excessive paging activity.

In another example, these memory-related views did in fact reveal a number of actual memory leaks in "F", a large Ada application. Due to the visual nature of these views, it was immediately apparent that particular leaks were flooding the address space (which was bleeding full of the color of the allocators in question) and hence required immediate attention. It was just as apparent that other leaks were inconsequential and could be ignored until after a rapidly approaching deadline. This is something which would not be readily apparent from the textual report of conventional special-purpose memory leak detectors.

4.3 Views of Hardware Activity and Source Progress

Finally, in an example involving "T", a computation-intensive scientific application, a view showing which loop of a program was active over time, in conjunction with a view of hardware performance statistics over time, highlighted opportunities for significant improvements in performance.

The program loop view is simply the application phase view described earlier, with color used to indicate which program loop is active at any instant in time (rather than which arbitrary user-defined phase is active). (Figure 6 shows this view in the window titled "AixPhase | ColorStrip".) The hardware performance view consists of a stack of linegraphs growing to the right over time. (Figure 6 shows this view in the window titled "RS2Pmc | Scale | LineGraph".) Hardware-level information, as discussed in Sections 2 and 3, is sampled at loop boundaries and plotted on the various graphs. In this case, the two views showed the same time span and were aligned for purposes of correlation and navigation.

These views allowed programmers to easily identify the longer-running loops and to correlate execution of a particular loop with a dramatic decrease in MFLOPS. The hardware view showed that the loop was not cache-limited and was not a fixed point loop, yet one floating point unit was seldom busy, while the other was extremely busy but completing very few instructions. To understand the behavior of this particular loop, a number of additional views were opened to show the program source.

Each source view highlights a line of source at the beginning of the major loop currently being executed. One of the views is, in effect, a "very high altitude" view of the source (as in [2]), in which the entire source of the program fits within the single window. Although the code is illegible due to the "very small font", the overall structure of the program is apparent, and the overall progress of the application can be tracked easily. The code in the second view of the source is legible, but the view can only show a page of source at a time and must be scrolled in order to view different parts of the program. (These two source views are shown side by side at the left of Figure 6, beneath the PV control panel.) The third view shows the assembly language source, as generated by the compiler, with the same form of highlighting as the other two source views. (In figure 6, this view is hidden behind the other windows.)

For application "T", glancing at the source views confirmed that, for the loop in question, a divide instruction was in fact causing one floating point unit to remain fully busy while not completing very many instructions. The assembly view showed that the reason for the second floating point unit not even keeping busy was an unnecessary dependence in the code. Using these views for feedback, the programmer was able to experiment rapidly with manual source transformations, and ultimately to achieve a 12% improvement in the performance of application "T".

Overall, through experience with PV in these situations and many others, the visualization capabilities proposed above have proven tremendously effective for debugging and tuning, often in cases where traditional methods have failed.

5 Future Research

The user interface is bound to be a severe limitation of *any* current software visualization system. Typical displays of software are crude approximations, at best, to the elaborate mental images that most programmers have of the software systems they are developing. Opening, closing, and aligning windows on a relatively small 2-dimensional screen is a cumbersome means of manipulating a few

small windows onto an elaborate conceptual world.

With the advent of sufficiently powerful virtual reality technology, a far more effective facility for software visualization could be achieved by mapping multiple-layer software systems onto expansive 3-dimensional terrains, and providing more direct means for traversal. Traversal could involve high-level passes over the terrain to obtain an overview, and descent to lower-levels over regions of interest for more detailed views. The system could also provide the ability to maintain a number of distinct perspectives onto the terrain. The panorama could include both representations of the software entities themselves, as well as derived information such as performance measurements, and more abstract representations of the entities and the progress of their computation.

6 Related Work

The notion of program visualization per se [15] [18] first appeared in the literature more than ten years ago [4]. Much of the initial work in program visualization, and many recent efforts, are concerned solely with the static structure of a program. They do not consider dynamics of program behavior at all.

Algorithm animation work [1] [17] has focused strictly on small algorithms, rather than on actual behavior of large applications or on all of the layers of large underlying systems. Further, algorithm animations often require large amounts of time to construct (days, weeks or even months). This is acceptable in a teaching environment, where the animations will be used repeatedly on successive generations of students, but is unacceptable in a production software development environment where it is critical that a tool can be applied readily to problems as they arise.

Recently, there has been much work in the area of program visualization for parallel systems [9] This work has in fact been concerned with dynamics, but much of it has been confined to communication or other aspects of parallelism. Little consideration has been given to displaying other aspects of system behavior. PIE [11] shows system-level activity over time, but its displays are limited primarily to context switching. Other system-level activity and activity from the application and other levels of the system are not displayed simultaneously for correlation.

The IPS-2 performance measurement system for parallel and distributed programs [5] [14] does integrate both application and system based metrics. However, system metrics are dealt with strictly in the form of "external time histograms", each describing the value of a single performance metric over time, as opposed to more general event data. Thus, where non-application data are concerned, IPS-2 is limited to strictly numeric presentations, such as tables and linegraphs. Dynamic animated displays of behavior, such as those showing system activity over time, or memory state as it evolves, are not possible with IPS-2. Program hierarchy displays are used primarily only for showing the overall structure of an application, or for specifying the program components for which performance measurements are to be presented.

Some vendors provide general facilities for tracing the system requests made by a given process. However, these facilities tend to apply to a single process rather than the system as a whole, and hence are not useful for showing the interaction between a process and its surrounding environment. Furthermore, these facilities tend to have very high overheads.

Profiling tools, such as the UNIX utilities "prof" and "gprof", have existed for some time, but these utilities simply show cumulative execution time, at the end of a run, on a function by function basis.

A number of workstation vendors have recently extended basic profiling facilities or debuggers by adding views to show time consumption and other resource utilization graphically. Many of these tools now report utilization with granularity as fine as a source line, and many allow sampling

during experiments which can cover some part of a run rather than just an entire run. None of these tools, however, supports the notion of general visual inspection of continuous behavior and system dynamics at multiple levels within a system.

Some debuggers are now including views of behavior in the memory arena, but none of these tools provides the power and generality of PV.

The power of PV, and its novelty, lie in its combination of a number of important properties. PV provides *both* quantitative and animated displays, and it presents information from *multiple* layers of a program and its underlying system. Further, PV facilitates *correlation* and *coordinated navigation* of the information displayed in its various views. Finally, PV presents views which address important concerns for software behavior on mainstream *workstation* systems, not just clusters or parallel machines. PV embodies all of these capabilities and it provides effective industrial-strength support of *large-scale* applications (even hundreds of megabytes of address space and hundreds of thousands of lines of code).

7 Conclusion

In production settings, over a wide range of complex applications, PV has proven invaluable in uncovering the nature and causes of program failures. Developers facing serious performance problems and imminent deadlines have found it worthwhile to invest time to connect PV to their application, and to run and inspect visualization displays.

Experience with PV indicates that concurrent visual presentation of behavior from many layers, including the program itself, user-level libraries, the operating system, and the hardware, as this behavior unfolds over time, is essential for understanding, debugging, and tuning realistically complex applications. Systems that facilitate visual correlation of such information, and that provide coordinated navigation of multi-layer displays, constitute an extremely powerful mechanism for exploring application behavior.

Acknowledgments

Heartfelt thanks to Keith Shields, Barbara Walters, and Christina Meyerson for hard work in the trenches, and to Fran Allen and Emily Plachy for unwavering support.

References

[1] M. Brown "Exploring Algorithms Using Balsa-II", IEEE Computer 21(5), pp. 14-36.

[2] S.G. Eick, J.L. Steffen, and E.E. Sumner, Jr., "Seesoft—A Tool For Visualizing Line Oriented Software Statistics", IEEE Transactions on Software Engineering 18(11), Nov. 1992, pp. 957-968.

[3] M.T. Heath and J.A. Etheridge "Visualizing the Performance of Parallel Programs", IEEE Software 8(5), Sep. 1991, pp. 29-39.

[4] C.F. Herot, G.P. Brown, R.T. Carling, M. Friedell, D. Kramlich, and R.M. Baecker "An Integrated Environment for Program Visualization", Automated Tools for Information Systems Design, H.-J. Schneider and A. J. Wasserman eds., North-Holland Publishing Company, 1982, pp. 237-259.

[5] J.K. Hollingsworth, R.B. Irvin, and B.P. Miller "The Integration of Application and System Based Metrics in a Parallel Program Performance Tool", Proc. Third Symposium on Principles and Practice of Parallel Programming, SIGPLAN Notices 26(7), July 1991, pp. 189-200.

[6] IBM Corporation, "AIX Version 3.1 for RISC System/6000 Performance Monitoring and Tuning Guide", IBM Corporation, order number SC23-2365-00.

[7] D.N. Kimelman and T.A. Ngo "The RP3 Program Visualization Environment", The IBM Journal of Research and Development 35(6), Nov. 1991.

[8] D.N. Kimelman and B.S. Rosenburg "Program Visualization for Implementation Performance Analysis", IBM Research Technical Report, October 1993.

[9] E. Kraemer and J. Stasko "The Visualization of Parallel Systems: An Overview", Journal of Parallel and Distributed Computing 18(2), 1993, pp. 105-117.

[10] T.J. LeBlanc, J.M. Mellor-Crummey, and R.J. Fowler "Analyzing Parallel Program Execution Using Multiple Views", Journal of Parallel and Distributed Computing 9(2), Jun. 1990, pp. 203-217.

[11] T. Lehr, Z. Segall, D.F. Vrsalovic, E. Caplan, A.L. Chung, and C.E. Fineman, "Visualizing Performance Debugging", IEEE Computer 22(10), Oct. 1989, pp. 38-51.

[12] A.D. Malony, D.H. Hammerslag, and D.J. Jablonowski, "Traceview: A Trace Visualization Tool", IEEE Software 8(5), Sep. 1991, pp. 19-28.

[13] A.D. Malony and D.A. Reed "Visualizing Parallel Computer System Performance", CSRD Report No. 812, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, May 1988.

[14] B.P. Miller, M. Clark, J. Hollingsworth, S. Kierstead, S. Lim, and T. Torzewski "IPS-2: The Second Generation of a Parallel Program Measurement System", IEEE Transactions on Parallel and Distributed Systems 1(2), Apr. 1990, pp. 206-217.

[15] B.A. Price, R.M. Baecker, and I.S. Small "A Principled Taxonomy of Software Visualization", Journal of Visual Languages and Computing 4(3), 1993, pp. 211-266.

[16] D.A. Reed, D.R. Olson, R.A. Aydt, T.M. Madhyastha, T. Birkett, D.W. Jensen, B.A.A. Nazief, and B.K. Totty, "Scalable Performance Environments for Parallel Systems", University of Illinois Technical Report UIUCDCS-R-91-1673, Mar. 1991.

[17] J. Stasko "TANGO: A Framework and System for Algorithm Animation", IEEE Computer 23(9), pp. 27-39.

[18] J. Stasko and C. Patterson "Understanding and Characterizing Software Visualization Systems", Proc. 1992 IEEE Workshop on Visual Languages, Sep. 1992, pp. 3-10.