

Integrating Real-Time and Partial-Order Information in Event-Data Displays

David J. Taylor Michael H. Coffin
Department of Computer Science, University of Waterloo

Abstract

The events occurring in the execution of a distributed or parallel application are related by a partial, rather than a total, order. We have developed prototype software that collects such events during program execution and produces a graphical display consistent with the partial order. Such a display can be very helpful in understanding and debugging distributed and parallel applications. However, using only partial-order information does not allow the performance characteristics of an application to be understood. Integrating real-time information with the partial order can provide a display that is useful for understanding both functional and performance aspects of the application. An algorithm is required to adjust the collected real-time information, to ensure that real times are consistent with the partial order. Lamport's clock algorithm provides such an adjustment, but can significantly distort the real-time values. It was necessary to develop a more complex algorithm, using the same basic principles, that minimises such distortions. We have extended existing prototype software for displaying event data, so that either a purely partial-order display or a real-time display may be obtained. The real-time facilities can be used in multiple target environments, such as OSF/DCE, Hermes, and SR.

1 Introduction

It is important to be able to understand the behaviour of a distributed or parallel application that is being developed. For many purposes, a display that shows the interaction between the processes or threads of an application can be a very useful aid to understanding. We had previously developed a prototype tool that allowed event data to be collected from an application and displayed according to the intrinsic partial order of the events. The resulting displays were useful for understanding the functional behaviour of an application but provided rather little information about performance characteristics.

Although a purely partial-order display allows some performance problems to be detected, such as an unexpectedly large number of calls from a particular client to a particular server, most performance problems can be detected only by using explicit real-time information. One approach would be to build a display based purely on the real times at which events occurred, but we wanted to retain the advantages of the existing partial-order display and also wanted to avoid display anomalies that might result from non-synchronised clocks. Thus, we chose to integrate real-time informa-

The IBM contact for this paper is Patrick Finnigan, Client Server Enabling Tools, Application Development Technology Centre, IBM Canada Ltd., Mail Stop 3P, 1150 Eglinton Avenue East, North York, Ontario M3C 1H7

tion with the existing methodology for building partial-order displays, rather than building a completely different display. This also allowed us to continue using the existing facility for process clustering and should allow use of future facilities for event abstraction, both of which can be used to eliminate currently irrelevant detail from the display.

The remainder of the paper is organised as follows. Section 2 contains a brief description of the work on which the real-time-display facilities are built, including the existing debugger prototype, real-time display facilities developed elsewhere, and algorithms for clock synchronisation. Section 3 describes the algorithm used to adjust real times to make them consistent with the partial order. Section 4 describes the construction and scrolling of real-time displays using the adjusted real times. Finally, Section 5 provides a summary and conclusions and suggests some possibilities for further work in the area.

2 Previous Work

Previous papers have described the basic event-collection and display techniques used in our partial prototype of a distributed debugger [11] and have described the use of process clustering to eliminate unwanted detail from displays [12]. This section briefly summarises the work discussed in those papers and then discusses some previous work in providing real-time displays showing the execution of parallel and distributed applications. Finally, a brief review of clock synchronisation is presented.

The basic concept behind our existing prototype is that “important” events are collected as an application executes and are then displayed, using a vertical line to represent each process (or thread or ...), placing a symbol representing each event on the appropriate line, and connecting the symbols for events representing process interactions. The “importance” of events is partly a matter of subjective judgment, but the set of collected events must include all those representing interactions between processes, such as message sends and receives. Additional events may be included if they are likely to provide useful information in

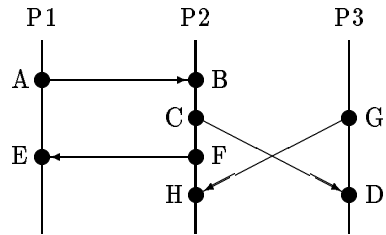


Figure 1: Simple example of an event diagram

the resulting display. The local execution of each process and the inter-process events induce a partial order on the complete set of events. The display is drawn so that it is consistent with that partial order. That is, if event A precedes event B in the partial order, A will be placed at a higher display position than B.

A simple example of an event diagram (which we have used previously [11]), containing three processes and eight events, is shown in Figure 1. In the figure, the horizontal connecting lines represent synchronous interactions between processes and the sloping connecting lines represent asynchronous interactions.

Such a diagram is, we would argue, exactly what is needed to determine whether process interactions are proceeding correctly; a conventional sequential debugger should be used to examine the internal activity of any individual process. However, neither the given diagram nor a sequential debugger provides much help in determining whether there is a performance problem. For example, it could be that some of the inter-event spaces represent very large amounts of computation and some represent only a few machine instructions.

The prototype provides many features needed to make such displays useful. One of the most important is an ability to scroll backward and forward in time, to view different parts of an execution history. Because the display is based on a partial order, a scrollbar does not suit this form of scrolling, so the scrolling is based on selecting an event and dragging it to the top or bottom of the display area. Other features include explicit display of precedence relationships, providing further information about displayed events, and drawing horizontal rather than vertical displays for those

who prefer that orientation. In addition, there is the important clustering facility mentioned above.

The prototype was carefully designed to limit the code that was dependent on properties of the target environment being debugged. The prototype originally worked with Hermes [10], but several other target environments are now supported: Concert/C [14], OSF/DCE [7], SR [1], the μ System [2], and the debugger itself. (The debugger is organised internally as multiple processes and has been instrumented so that it can be used on itself.) The extensions that have been made to the debugger for real-time support are, in principle, available in all these target environments. Since, obviously, the instrumentation in the target environment also needs to be extended to collect the real-time value, we can presently only use real time in the Hermes and self-debug versions of the debugger and will use Hermes as a source of examples for this paper.

There are several examples of systems that display information about the execution history of a distributed or parallel system with real time used as one axis of the display. Many of these are for parallel systems, which have the simplifying advantage of a common clock to provide a consistent indexing on the real-time axis. In a paper summarising a variety of techniques for displaying information about the execution of parallel programs [8], such process-time diagrams are indicated as one of a handful of commonly used techniques. Although there are many variations of detail in the displays produced, some look very much like our existing partial-order displays [3], except that one axis represents real time. Thus, there is a sufficient compatibility between the displays so that integrating real time with the event partial order appeared to be a reasonable approach. However, in a distributed system, clocks are not perfectly synchronised and can yield event times inconsistent with the partial order.

A great deal of work has been done on synchronising clocks in distributed systems. Lamport [5] describes an algorithm for assigning times to events in a distributed computation so that the times are consistent with the partial order. This work is fundamental to the approach taken here, however, used by itself, this

technique can result in substantial distortions in event times since event times are moved forward only when necessary to be consistent with the partial order.

Lamport and Melliar-Smith [6] discuss the problem of synchronising clocks in a distributed system in the presence of faults. Many authors have since added to this line of research [13, for example]. However, this work is not directly applicable to the problem of adjusting times in the prototype since it involves sending extra messages, often broadcasts. Furthermore, the clock synchronisation achieved by such methods is not necessarily consistent with the partial order of application events.

Heath and Etheridge [4] describe ParaGraph, a system in some ways similar to our prototype. The authors note the possibility of observing “tachyons”—messages that travel backward in time. To avoid this problem, they base all communication on the PICL message-passing library, which, in addition to providing portability, performs clock synchronisation. Since our prototype debugger is designed to be re-targetable, and since most potential target systems do not provide clock synchronisation that is guaranteed to avoid such anomalies, this approach was not possible.

3 Time-Adjustment Algorithms

As we began to add real-time information to the prototype displays, it became clear that the simple, largely implicit, abstraction used to represent synchronous communication would be a source of difficulty. Synchronous communication has the property that the two paired events conceptually happen simultaneously but, of course, they do not happen at precisely the same instant of real time. To avoid distorting the real-time display but still represent synchronous communication as synchronous, it was necessary to define some additional events. Rather than simply having the pairs call/receive-call and return/receive-return to represent a remote procedure call, we now use two successive events to represent the call and two successive events to represent the receive-return. In each case, one of these events

pairs synchronously with an event in the other process and the other represents the actual activity at this process. For similar reasons, if a receive must block waiting for a message to arrive, it is recorded as two events, a receive-blocked and a receive-complete.

In most cases, the split events have real significance for the user. An example is shown in Figure 2, in which it is assumed that P2 blocks before receiving the call. In this case, the call will not block, so a real-time display will show a very short interval between the call and call-complete events. In general, there will be significant information in one but not both of the intervals from call to call-complete and receive-blocked to receive-call. As indicated above, if the receiving process does not block, the receive-block event is simply omitted. Because the blocking of a call is determined at the remote location, it is difficult to omit the initial call event in cases that the call does not block. The only interval that conveys little or no real information is that between receive-return and receive-complete, since it represents the delay between the issuance of the return and its arrival at the calling process. For most purposes, this is just part of the (long) delay from call-complete to completion of the return operation.

When an event is generated by the target program, the clock time, as given by the clock of the local processor, is recorded in the event. If all clocks on the system were perfectly synchronised, this raw clock time could be used directly in the display algorithm. However, processor clocks are, in general, not perfectly synchronised. If raw times are not corrected, they can result in distorted, or even inconsistent, displays. For example, the raw time at which a message is received may be less than the raw time it was sent. To avoid inconsistencies and distortions, the raw times must be corrected.

The problem of correcting raw times is, in principle, similar to the problem of synchronizing clocks. However, standard clock-synchronisation algorithms are not directly applicable, for the following reasons. First, the debugger operates passively in order to minimise the probe effect. Thus time-synchronisation protocols are not possible; the time correction must be done with what infor-

mation is already available in the event stream. Second, most clock-synchronisation algorithms attempt to minimise the maximum difference among a collection of clocks. For a real-time display, different criteria are important: it is imperative to avoid assigning times that are inconsistent with the partial order, and it is highly desirable to minimise the distortion of short inter-event intervals.

Suppose that a call is made from one processor to another, and that the call event is recorded with raw time t_c on one processor, and that the corresponding receive-call event is recorded with raw time t_r , as measured on the other processor. (The return and receive-complete event pairs are treated similarly.) The apparent latency L of the message is

$$L = t_r - t_s$$

If the apparent latency is larger than the minimum possible latency (which depends on the hardware and software connecting the two computers, but is always positive), then little or no information about the errors of the clocks can be inferred. If, however, the apparent latency is less than the minimum possible latency, then it must be the case that the clock of the calling process is ahead of the clock of the other process. The actual latency is

$$L = (t_r + e_r) - (t_s + e_s)$$

where e_r and e_s are the errors in the raw times. This observation can be used to estimate the error of the two clocks if some assumptions are made.

Assume that, for each clock, the errors are normally distributed with a fixed mean and variance. The equation above implies that

$$e_r - e_s = L - (t_r - t_s)$$

The true value of L is unknown, but since the observed latency is less than the minimum, the actual latency is probably near the minimum. If it is estimated that $L \approx L_{min}$, then

$$e_r - e_s \approx L_{min} - (t_r - t_s)$$

where all quantities on the right-hand side are known. It remains to apportion the net error (the right-hand side) to e_r and e_s .

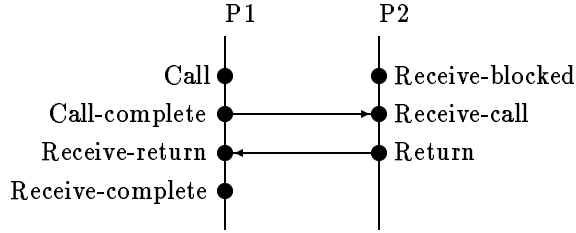


Figure 2: Example of additional events

If e_r and e_s are independent random variables and drawn from normal distributions, the maximum-likelihood estimate of e_r and e_s is that they are an equal number of standard deviations from the means of their respective distributions. Thus they must satisfy the following equations, for some x

$$e_s = \bar{e}_s - x\sigma_s \quad (1)$$

$$e_r = \bar{e}_r + x\sigma_r, \quad (2)$$

where \bar{e}_s and σ_s are the mean and standard deviation of the errors from the clock of the receiving process, and \bar{e}_r and σ_r are the mean and standard deviation of the errors from the clock of the receiving process. These two equations, in addition to the previous one, are enough to approximate e_s and e_r .

The computation described above requires the mean and standard deviation of the two distributions. However, given a number of samples, an estimate of these parameters can be calculated. In practice, this means keeping, for each processor p in the system, the following quantities:

- n , the number of samples collected for processor p ,
- $\sum e_p$, the sum of the samples, and
- $\sum e_p^2$, the sum of squares of the samples.

These are sufficient to calculate the sample mean and standard deviation by well-known formulae [9].

This algorithm results in an estimate of the clock error that improves as interactions between processors take place. When an event arrives at the debugger prototype, the current estimated error for its processor is used to correct the raw real-time information. In practice,

this means that, if clocks are poorly synchronised, inter-event durations are distorted at the beginning of the execution, but become less distorted later in the execution. For implementation reasons, the prototype does not revise event times based on improved estimates of the error.

It sometimes happens that the correction arrived at by the procedure described above can result in an inconsistent display. This is because a negative correction can potentially place the real time of an event before one or more of its predecessors in the same trace. In this case, the time of the event is corrected to be slightly greater than the time of the most recent event in the same trace, and the time of the partner event is moved forward by the same amount.

This computation makes an important assumption that may not always hold, namely that the distribution of errors for a clock is static and does not drift over time. This assumption is usually a reasonable approximation for programs that execute for short periods, but more sophisticated algorithms may be required for long-running programs. Another important assumption is that the clock-synchronisation algorithm, if one is being used, does not introduce sudden changes in clock times. A clock-synchronisation algorithm that adjusts rates to be nearly equal, on the other hand, is helpful.

4 Real-Time Display

In most respects, creating a display based on real time is much simpler than creating a display based on a partial order. The essential idea is simply that each event is placed at a position corresponding to its real time, in one

direction, and corresponding to the process it comes from, in the other direction. Given that events with adjusted real times are supplied to the display algorithm, synchronous pairs of events are automatically positioned correctly with respect to each other. There are a few problems: proper handling of clusters, overcrowded displays, and large gaps in the display.

As discussed previously, one advantage of integrating real-time facilities with the existing prototype is that existing facilities for clustering can be used with real-time displays. With either a partial-order or a real-time display, it can be useful to remove from the display events that are internal to a set of processes (a cluster). Eliminating events that are not currently of interest can be important if an application includes many processes, in order to allow easy examination of that part of the application that is currently of interest.

Two issues arise because of clustering. One is simply that the order in which events are placed on the display becomes important. The obvious choice of placing all the events for process 1, all the events for process 2, and so on, does not work with clusters. The problem is that events from several processes will be displayed on a single display "trace", and these will not arrive sequentially if each process is handled separately. The first problem would be quite minor if not for the second problem, which is that in some cases the events displayed for a cluster may not be precedence-related. When such situations occur, in order to properly reflect the partial order, the trace line used to display the cluster is split into two trace lines (or more, as required), with events distributed among the trace lines such that any two events on the same line are precedence-related. Making insertions into the middle of such a collection and then rearranging the displayed events is practically impossible, so the display must be built by adding only onto the "ends" of a displayed cluster. As a result, the most practical technique for building the real-time display is to scan all traces in parallel, placing all events for each value of real time, then proceeding to the next value, rather than working process by process.

Another possible problem is an overcrowded display. In a partial-order display, the debug-

ger works internally with a notion of display rows and will place at most one event per process per display row. In a real-time display, the user can specify the scale of the display (currently the specification is given in microseconds per pixel) and it is possible to specify a scale such that the entire execution history of a large application will be displayed. Such a display might or might not be legible, but the practical problem is that the debugger is designed so that all displayed events are kept in main-storage (actually, virtual-storage) data structures and it is not reasonable to pick up the entire content of a large event-data file and put it into such data structures. Thus, the program sets a limit to the "crowding" allowed on the display. Since, even when building a real-time display, the program uses a notion of display rows internally, the limit is specified in number of events per display row rather than for the complete display. If the limit is exceeded, the scale is automatically adjusted to create a display that does not exceed the limit.

At present, the limit is set at twice the number of processes, but is never less than 20 even if there are fewer than 10 processes. Since, in any small section of execution history, process behaviour is likely to be highly non-uniform, this will likely mean no events from most processes and many events from a few processes, unless there are very few processes. In practice, the present limit seems to allow construction of displays that are so dense they are illegible, so a higher limit is likely not useful.

The final problem is one that we had not initially anticipated. It is not unusual for quite large real-time gaps to occur in an execution history. As a result, a display may be created in which there are several events very close together, a big gap, then several more events very close together. If the scale factor is changed to allow individual events to be distinguished, the gap becomes so large that it is not possible to have events from both sides of the gap on-screen simultaneously. Of course, such a problem cannot occur in a purely partial-order display and we failed to anticipate that there would be such large variation in inter-event times in a real-time display. The solution is to "squeeze" the gaps. Although we have not yet implemented the solution, our intention is

to note a situation in which some number (say, three) of consecutive display rows will be vacant and then compress the gap to a small number of display rows (say, two), with a saw-tooth line drawn across the display to indicate a discontinuity in the real-time axis. Fortunately, the display construction proceeds sequentially through values of real time, so detecting and compensating for such large gaps will be easy to do.

In Section 2, when discussing scrolling of a partial-order display, it was implicitly suggested that a scrollbar could be used to scroll a real-time display. At this point, we have chosen not to implement such a scrollbar, but rather to perform scrolling in the same fashion for both partial-order and real-time displays. Thus, to move backward or forward in real time, the user selects an event and requests that the debugger drag the event to the top or bottom of the display. All the event-selection facilities provided for partial-order scrolling are then available for real-time scrolling: selection of an event in the current display, selection of a specific event number from a particular process, selection of an event to which a label was previously attached, etc.

Figure 3 illustrates a partial-order display showing processes from two Hermes interpreters. The processes near the top of the display, down to `ac_none`, are from one interpreter and the remaining processes are from the other interpreter. Figure 4 shows the corresponding real-time display. The events in the upper-left hand quadrant of the partial-order display do not appear in the real-time display because, in real time, they occurred much earlier than the other events. There is a collection of events toward the right side of the real-time display that are very hard to distinguish; however, we could change the scale of the display to examine them more closely.

5 Conclusions and Further Work

In this paper, we have described the extension of an event-display facility based purely on event partial order to handle real-time information as well. Having worked with both types of

displays, we strongly believe that each has appropriate uses and that neither could be effectively used as a replacement for the other. The adjustment of real-time values so that they are consistent with the partial order, while trying to minimise real-time distortion, is not a simple problem, but our initial solution appears to provide reasonable behaviour. We may choose to implement and test alternative algorithms, but before doing so, it would be prudent to characterise the “goodness” of such algorithms, so that there would be a quantitative basis for comparison. Fortunately, most other aspects of creating a real-time display were, in principle, straightforward, although substantial coding effort was required to modify the existing display data structures so that they could be used effectively in both partial-order and real-time display construction.

Clearly, several issues still need to be pursued. As mentioned above, the facility for eliminating large gaps from the display has not yet been implemented, but should be fairly easy to do. As well, the complete set of target environments for the debugger does not yet support real-time displays. The significant problem is not in returning a real time with each event, but rather the appropriate collection of the additional events needed to handle synchronous communication nicely in a real-time environment, as discussed in Section 3. In fact, we are in the process of modifying the implementation to eliminate the need to collect most of the additional events; instead they will be synthesized as the display is constructed. We also need to acquire more experience in using the real-time version of the prototype, preferably using it to do some actual performance tuning. Such experience will also likely suggest further useful additions to the facilities provided.

Acknowledgments

The work described in this paper began in the Shoshin project at the University of Waterloo and has more recently benefited from numerous discussions with members of the Centre for Advanced Studies, IBM Software Solutions Toronto Laboratory. The work was supported by NSERC, under Operating Grants

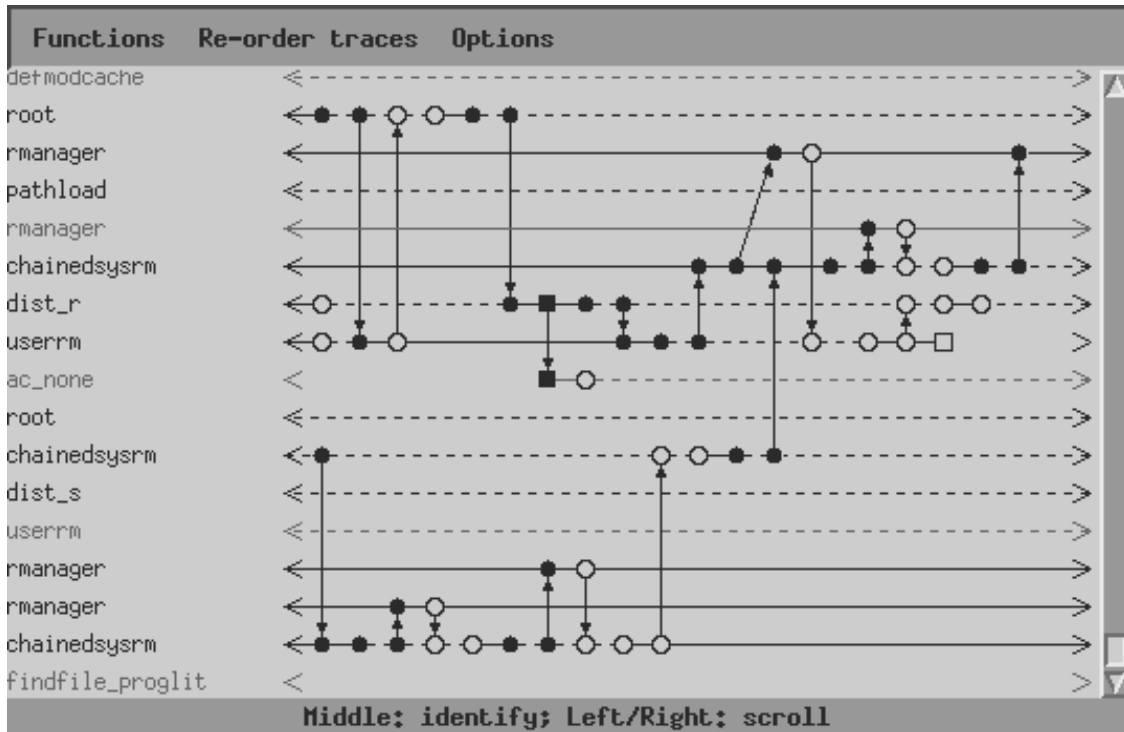


Figure 3: Example of a partial-order display

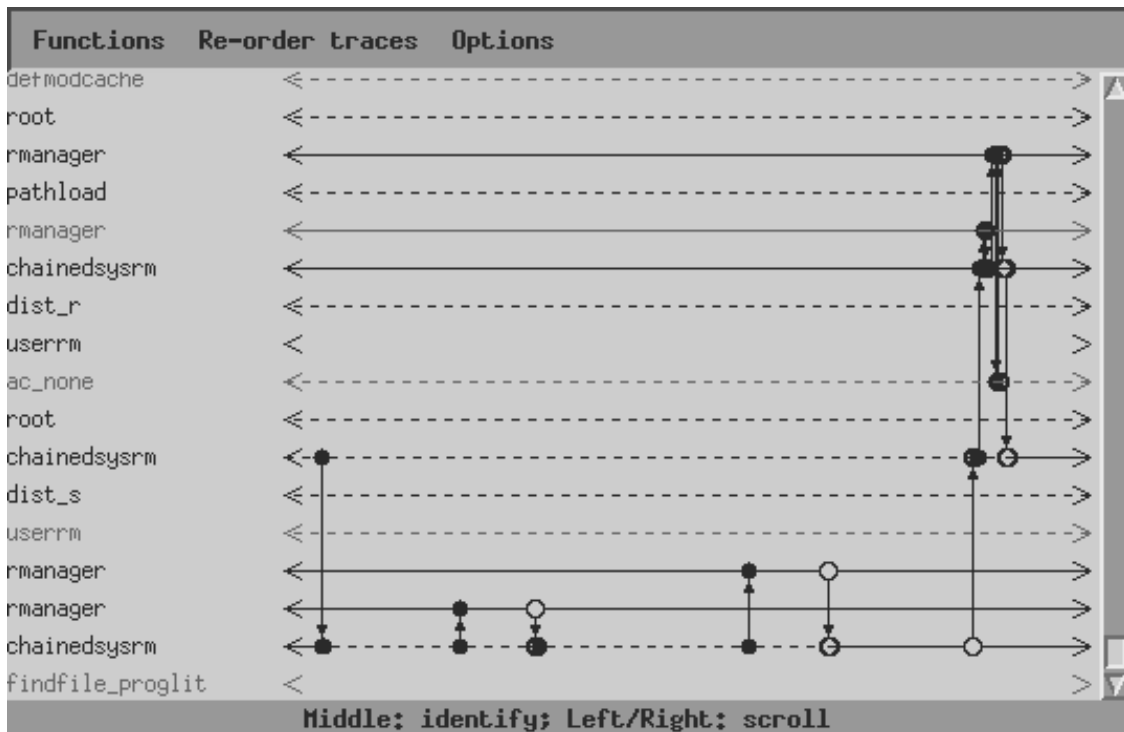


Figure 4: Corresponding real-time display

OGP00003078 and OGP0105585 and a Senior Industrial Fellowship, by the Information Technology Research Centre, and by IBM.

About the Authors

David Taylor is an Associate Professor of Computer Science at the University of Waterloo, where he has been a faculty member since 1977. His research interests include distributed-systems software and software fault tolerance. During the 1991-1992 academic year, he spent a sabbatical at the Centre for Advanced Studies, IBM Canada Ltd. Laboratory.

He can be reached at the address Department of Computer Science, University of Waterloo, Waterloo, Ontario N2L 3G1. His Internet address is dtaylor@uwaterloo.ca.

Michael Coffin is an Assistant Professor of Computer Science at the University of Waterloo, where he has been a faculty member since 1990. His research interests include distributed systems and programming languages.

He can be reached at the address Department of Computer Science, University of Waterloo, Waterloo, Ontario N2L 3G1. His Internet address is mhcoffin@uwaterloo.ca.

References

- [1] Gregory R. Andrews et al. An overview of the SR language and implementation. *ACM Transactions on Programming Languages and Systems*, 10(1):51-86, January 1988.
- [2] Peter A. Buhr and Richard A. Strooboscher. The μ System: Providing light-weight concurrency on shared-memory multiprocessor computers running UNIX. *Software—Practice and Experience*, 20(9):929-963, September 1990.
- [3] Robert J. Fowler, Thomas J. LeBlanc, and John M. Mellor-Crummey. An integrated approach to parallel program debugging and performance analysis in large-scale multiprocessors. In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, Madison, Wisconsin, May 5-6, 1988.
- [4] M.L. Heath and J.A. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, 8(5):29-39, September 1991.
- [5] Leslie Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558-565, 1978.
- [6] Leslie Lamport and P.M. Melliar-Smith. Synchronizing clocks in the presence of faults. *JACM*, 32(1):52-78, January 1985.
- [7] Open Software Foundation. *Introduction to OSF/DCE*. Prentice-Hall, 1993.
- [8] Cherri M. Pancake and Sue Utter. Models for visualization in parallel debuggers. In *Proceedings of Supercomputing '89*, pages 627-636, November 13-17, 1989.
- [9] George W. Snedecor and William G. Cochran. *Statistical Methods*. Iowa State University Press, eighth edition, 1989.
- [10] Robert E. Strom et al. *Hermes: A Language for Distributed Computing*. Prentice-Hall, Englewood Cliffs, New Jersey, 1991.
- [11] David J. Taylor. A prototype debugger for Hermes. In *Proceedings of the 1992 CAS Conference, Volume I*, pages 29-42, November 9-12, 1992.
- [12] David J. Taylor. The use of process clustering in distributed-system event displays. In *CASCON '93*, pages 505-512, October 24-28, 1993.
- [13] J.L. Welch and N. Lynch. A new fault-tolerant algorithm for clock synchronization. *Information and Computation*, 77(1):1-36, 1988.
- [14] Shaula Yemini et al. Concert: A high-level-language approach to heterogeneous distributed systems. In *Proceedings of the 9th International Conference on Distributed Computing Systems*, pages 162-171, June 5-9, 1989.