RC 21281(94784)14SEP98 Computer Science/Mathematics

Research Report

A Methodology For Performance Analysis Using System Event Plots

Ronald Mraz
IBM T.J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598
mraz@watson.ibm.com

Research Division
Almaden · Austin · Beijing · Haifa · T.J. Watson · Tokyo · Zurich

A Methodology for Performance Analysis Using System Event Plots

Ronald Mraz

IBM T. J. Watson Research Center
Yorktown Heights, NY 10598
mraz@watson.ibm.com

Abstract

This paper describes a methodology for analyzing complex computer systems performance. The charts generated by this technique provide the user with an intuitive feel for the system's operation and performance. Two types of Event Plots are described: traditional Impulse Plots and a generalized method termed Mraz Plots. Mraz Plots require more processing of sampled data but provide a more informative plot when the events are within an order of magnitude of the system's monitoring window. This is typically the case for complex multitasking, layered software systems. To illustrate the effectiveness of the generalized technique, we provide data from actual systems, and review detailed case studies where this methodology was employed to improve system performance.

1 Introduction

As computer system performance has improved, the complexity of tasks performed on the systems has increased to take advantage of the technology. Modern operating systems with internet connections employ multiple execution threads to support and service their interfaces. It is not feasible to trace each and every service (or daemon) that can impact the performance of a user's application. Therefore, a methodology is employed that can analyze the entire system at once. CPU Utilization Plots and Impulse Plots are the traditional ways that this analysis has been done. Although, these traditional methods can only provide meaningful data when the granularity of all events to be examined are small with respect to the examination window, and all event durations are within an order of magnitude in time.

Figure 1 provides an event plot of a small single threaded embedded operating system kernel termed NAP for "Naked Application" execution environment. This environment, which we used to characterize the PowerPC virtual memory subsystem, as shown in Reference [5], provides a low level operating system to run applications. The only application running for this experiment is the Event Plot monitor which records all system events. As we can see, the trace is void of activity except for the small events that occur 18 times a second. These are timing interrupts that can schedule other tasks for operation. The methodology demonstrates that NAP is a highly efficient environment for executing single in-line programs.

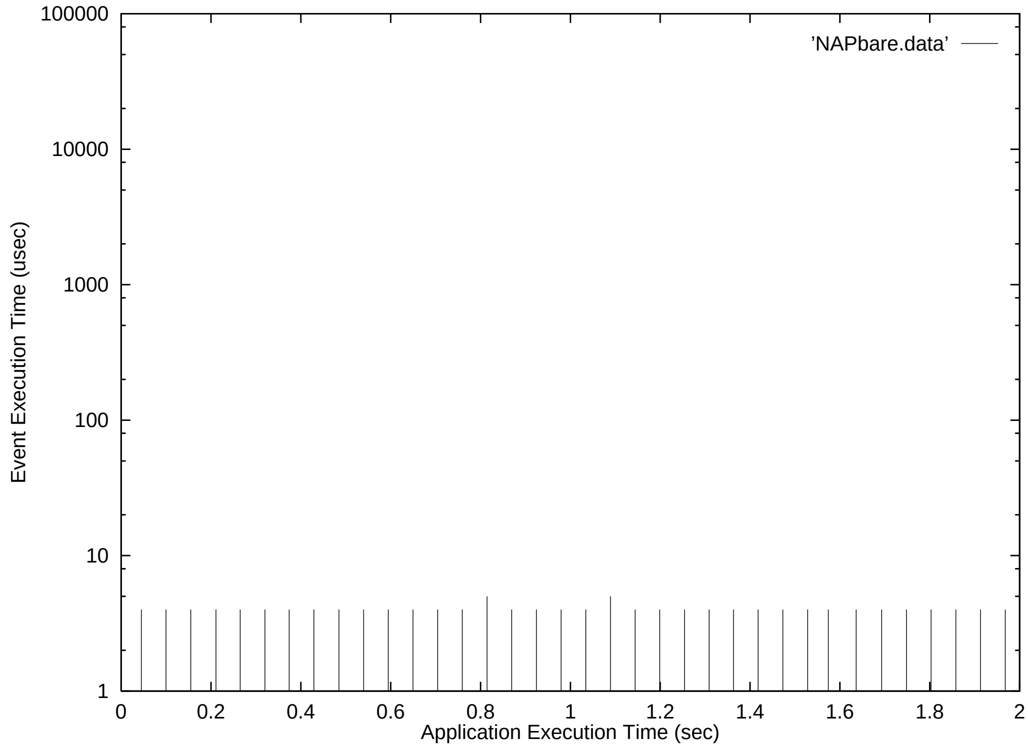

Figure 1: Plot of Events within an Embedded Kernel.

Modern operating systems are much more complex, since many threads of execution are scheduled concurrently. Figure 2 provides us with an Impulse Plot of all events within the IBM AIX [1] operating system. Most of the events are timer interrupts for scheduling system daemons and housekeeping. These occur 100 times a second and are around 35 microseconds in duration. When these interrupts require the processor to perform a task, we see these interrupts are in the 100 to 1000 microsecond range.

Additional complexity is added to the system through middleware components that might provide a database interface or internet services as well as a custom message passing solution. Finally, the user's application can have multiple threads of execution for efficiency and division of labor. As we have seen, the event duration of an embedded system can grow from a few microseconds to 100's or 1000's of microseconds. Middleware and application event durations can range from 1000's of microseconds to full seconds. Given that these events can span three or more orders of magnitude, we need to refine our tools to reflect the increasingly complex systems we are analyzing.

A Mraz Plot provides a generalized version of the Impulse Plot, as shown in Figure 6 where we have an event[1] that is over a quarter of second long. To illustrate the event duration, we have constructed a rectangle whose logarithmic height is equal to its linear width in time. If this was represented by a simple impulse, the observer would have to determine the event termination by other means of measurement. Impulse plots are, therefore, a special case of this method since all events much smaller than the observation window appear as impulses. It is these longer events of complex systems that require the rectangular representation for accurate interpretation.

This paper describes a methodology for examining and characterizing complex computer systems.

[1] This event is really made up of middleware and many other operating system interactions, but, to the user, it appears as one long event.

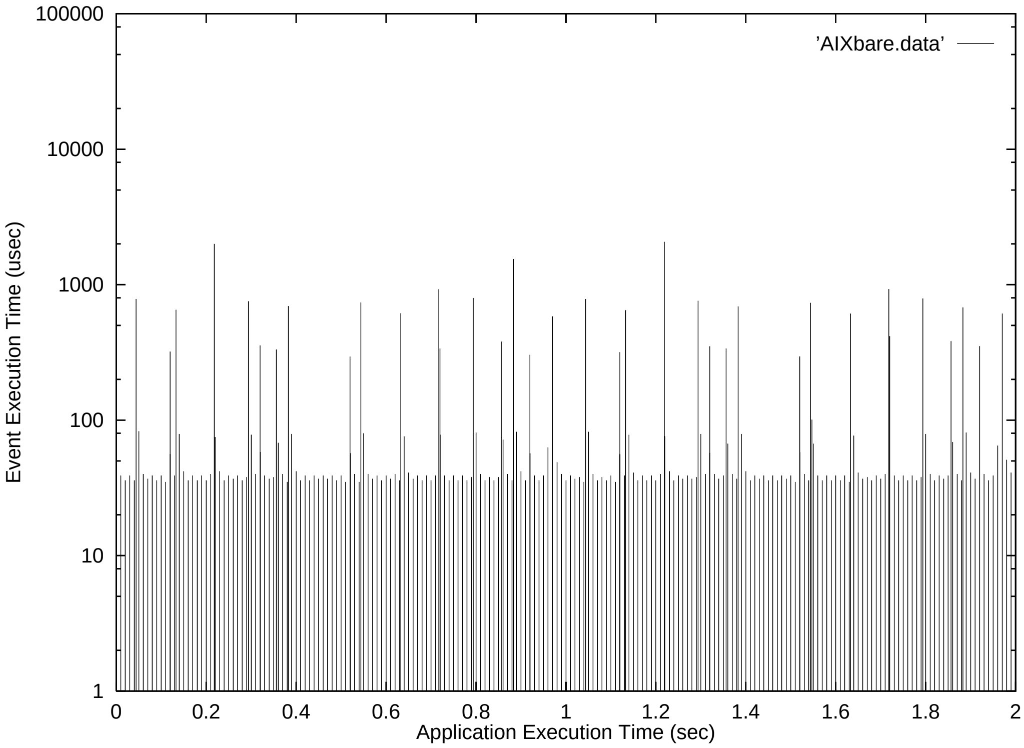

Figure 2: Plot of Events within a Modern Operating System.

The measurements do not require any modification of the user program[2] or source level debugging information. This methodology can aid the user in identifying components of the system that have high CPU utilization requirements as well as overall performance considerations.

1.1 Roadmap

This section provides an introduction and background to the work done. Section 2 provides a brief overview of the methodology and execution environment we have implemented. Section 3 provides a case study for examining the message passing middleware of a high performance parallel computer. Section 4 provides a case study for an analysis of a commercial TCP/IP based client/server backup program. Section 5 is a summary and directions for future work.

2 The Event Plot Methodology

There are two major steps to the generation of these plots. This first is collecting data in real-time while the subject application is running. The second is processing the analysis and display. We describe each separately.

2.1 Data Collection

The collection program is a program that loops for a definite period of time. This monitoring time is set to be longer then the system action analyzed, and is started prior to triggering the subject event.

[2] Although, timestamps within the source can provide additional insight for analysis.

The collection program loops continually and measures the difference in time from one iteration to the next. If this time exceeds a specific threshold, say 100 microseconds, some program event has interrupted our loop and we can now record it's time and duration.

During the event operation, the Collection Program should run at the lowest[3] possible priority. This will allow the program to witness all events within the machine occurring within the monitoring window. Below we have pseudo code that shows the operation of the collection program.

```
// pseudo C code for data collection
     set_priority(low);          // In AIX set priority to 119
     get_clock(&rtc1);           // read real time clock
     while (monitoring)          // loop endlessly
       {
       get_clock(&rtc2);         // read real time clock
       rtcd = rtc2 - rtc1;       // calculate diff from last reading
                                 // Were we delayed? If yes, record.
       if ( rtcd > usec_threshold)
           printf(``delay pf %d encountered prior to %d sec %d usec'',
              rtcd, rtc2.sec, rtc2.usec);
       rtc1 = rtc2;              // prepare for next iteration
       }
```

2.2 Data Processing

The data processing portion takes the raw output and creates a polygon placed at the start of the event whose x-axis width is the duration of the event. The y-axis height is also the duration of the event. When plotted on a log scale, the polygon appears as a tall rectangle. The following code generates the four X and Y coordinates from the Collected Data.

```
// pseudo C code for data processing
     Xb_old = 0.0;
     threshold = 100;  // minimum resolution usec
     EOF = (scanf (``%f %f, &time_sec, &delta_usec);
     while (!EOF) {         // loop for all data points
       Xb = time_sec;
       Y = delta_usec;
       printf(``%f %f\n'', Xb_old, threshold);
       printf(``%f %f\n'', Xa, threshold);
       printf(``%f %f\n'', Xa, Y);
       printf(``%f %f\n'', Xb, Y);
       Xb_old = Xb;
       }
```

3 Message Passing Case Study

This methodology was able to help characterize and identify all system components that contribute to optimizing a message passing system [3] for reduced variance in communication delays, Figure 3 provides the Event Plot[4] as seen by the user's application. The overall goal of this work was to

[3] But, above the System Idle Process.

[4] An Impulse Plot was used since the events are much smaller than the observers time scale.

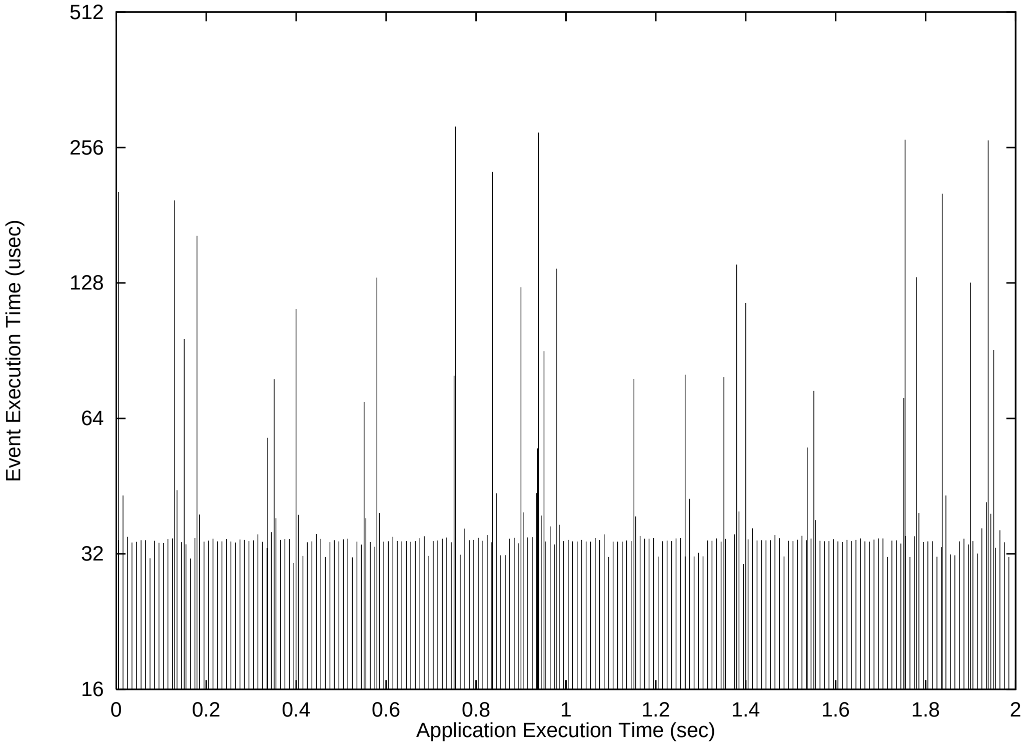

Figure 3: Event Plot of User Level View of the Message Passing System.

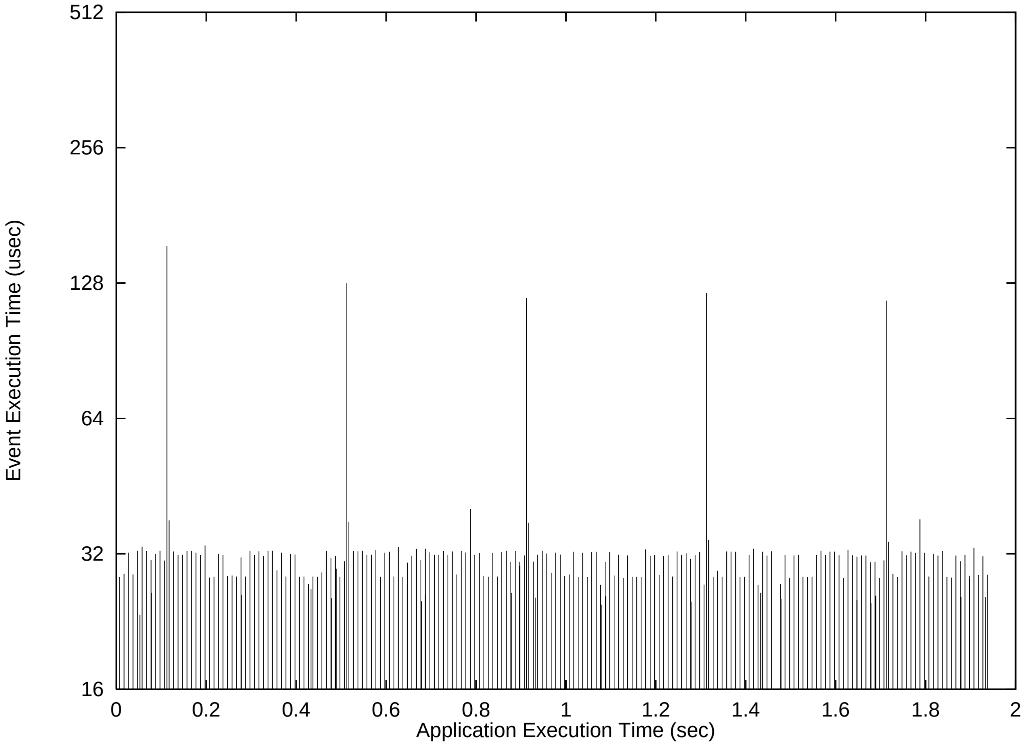

Figure 4: Event Plot of a Message Passing System.

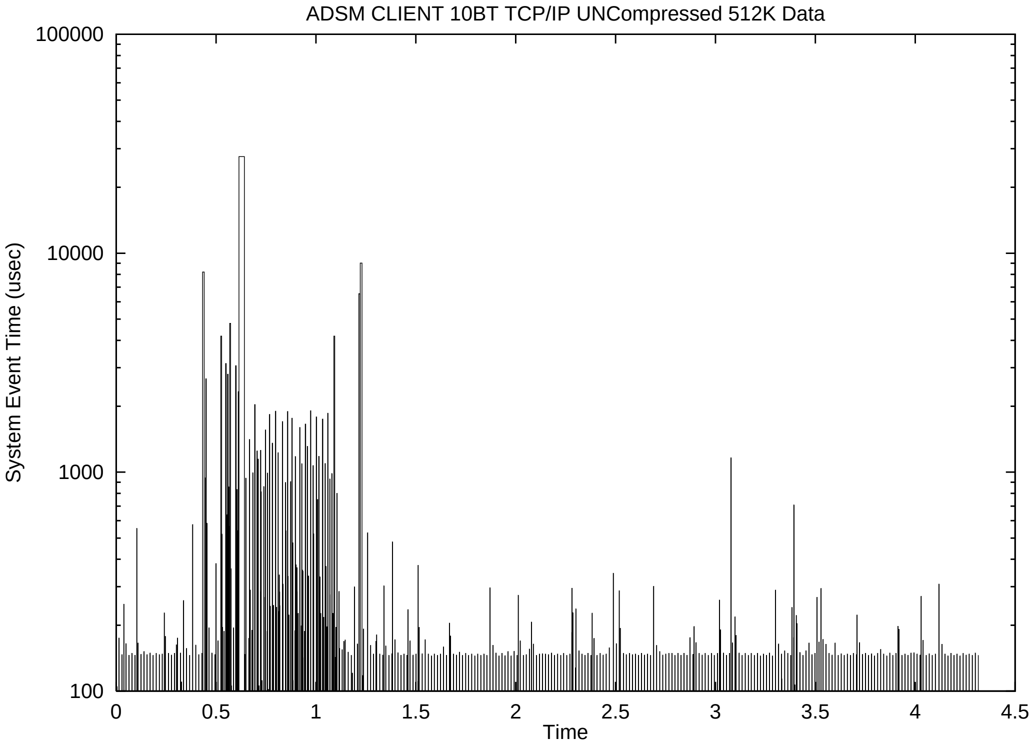

Figure 5: TCP/IP Backup Client - No Compression.

identify and characterize all ways to reduce the variance of user space messages. Each point to point message can be sent and received in under 30 microseconds. The variance results when one of these events delays the sending or processing of the message. Since these events are several times longer than the message transfer, variance can be significant.

We have also found that the message passing system utilized a keepalive daemon that itself contributed to message passing variance. This timer based service is characterized in Figure 4. Comparing Figure 4 with that of Figure 3 shows that these timer based events are present as well. Since this is an essential service to the message passing system, the application must yield to these events.

Therefore, the application can have a priority above all other services in the system but below that of the message passing daemon. It was found that by setting the application's priority slightly higher than the processes causing the longer events, we achieved nearly all the benefits of optimal execution. The primary difference was the variance caused by the message passing keepalive daemon. A detailed treatment is provided in [4].

4 Backup Client Case Study

To understand and enhance the performance of a backup client [2] in a networked backup program, one must consider several components that contribute to performance. Using this methodology, we found there are two competing effects in the system architecture, namely compression and network communications. Specifically, the more you compress, the better the network communication is, but, the more aggressive the compression algorithm, the slower the system becomes due to the time it takes to compress data in the software.

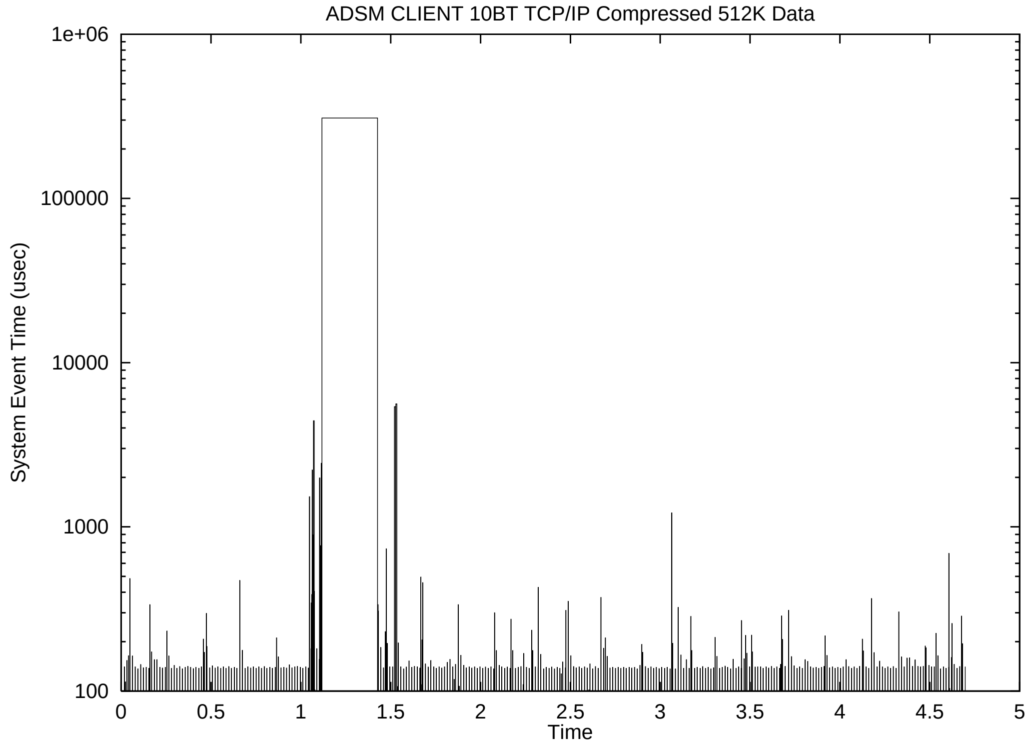

Figure 6: TCP/IP Backup Client - Compression Enabled.

Figures 5 and 6 represent Mraz Plots[5] of backing up a 512K byte file with compression and without compression enabled at the client. Analysis of Figure 5 shows the stages of the high function backup program. The initial query to the server occurs at 0.4 seconds. The servers response is received at 0.5 and the associated metadata for backup recovery is prepared. At 0.6 seconds, the large event is a memory copy of the file to a staging area to combine the metadata with the file and provide segmentation of the data for TCP/IP transfers. The TCP/IP transfer occurs from 0.6 seconds until 1.1 seconds. At 1.2 seconds the server responds that data has been committed to disk and the backup operation is complete.

Analysis of Figure 6 shows the stages of the high function backup program. The initial query to the server occurs after 1.0 seconds. The server's response is received at 1.1 seconds and the associated metadata for backup recovery is prepared. Continuing as one long event is the memory copy to the staging area and the compression of the data for TCP/IP transfer. Having compressed the file by 97 percent,[6] we find a small transfer of data prior to 1.5 seconds. At 1.5 seconds the server responds that data has been committed to disk and the backup operation is complete.

What these two graphs show are that the major contributors to CPU loading are compression and the memory copy of data to a staging area. From this study, work is proceeding to upgrade the software compression algorithm. Additionally, we find that the software compression algorithm can only compress data at rates of 2 Megabytes per second in a 233 Mhz PowerPC. This can be a potential bottleneck when archiving large amounts of data.

[5] We use the generalized plotting method since the event durations are on the order of the observers time window.

[6] The file consists of 512K bytes of replicated alphanumeric text. Very efficiently compressed by modern compression methods.

5 Summary

To analyze complex systems, we have generalized a methodology for performance analysis. We demonstrated the usefulness of this methodology over traditional methods with examples from a complex middleware message passing program and a high function, TCP/IP client/server backup program.

Impulse plots were used when the events themselves were much smaller than the time monitoring window. In our examples, we measured events on the order of 30-100 microseconds over a 2 second monitoring window. These events wre smaller than the event window by 2 orders of magnitude difference.

Mraz plots were used when events that we were measuring were within an order of magnitude of the monitoring window. In our backup client example, we are measuring events on the order of 0.5 seconds over a 5 second monitoring window or within the same order of magnitude difference.

Future work would includes migrating this analysis to PC-based systems including NT and Windows based client and servers.

Acknowledgements

The author wishes to express his gratitude to the many individuals who have helped provide resources and information critical to this work. To Jim Carlson, Robert Morris and Jai Mennon for their management support and guidance. To David Pease and Bernard Lopez of the Almaden Research Center for support and information regarding the backup software functions. To Prasad Rao at IBM Almaden Research who helped generate the backup client data and processing software.

To Brent Paulovics and Ron Ambrosio, who provided the essence for NAP from the much larger CPQ operating environment. To Peter Hochchild who developed the experimental message passing system and found similar plots useful in his own analysis. To Marc Snir, C. J. Tan, Gabby Silberman, Pratap Pattniak and others who supported and guided different aspects of this work.

About the Author

Ronald Mraz is a Research Staff Member at the IBM T. J. Watson Research Center, Yorktown Heights N.Y. He received his Ph.D. in Computer Engineering from Carnegie Mellon University, where he was supported through an IBM Fellowship. Dr. Mraz's research interests include computer architecture, real-time systems, operating systems, parallel and clustered systems.

References

[1] *AIX Version 3 for RISC System/6000 Technical Reference.* IBM Documnet Number SBOF-0539-00.

[2] *Using the UNIX ADSM Backup-Archive Clients.* IBM Documnet Number SH26-4052-00.

[3] Peter Hochchild. EUIH: An Experimental EUI Implementation Version 1.06.03. Technical report, IBM Research Report, 1993.

[4] Ronald Mraz. Reducing the Variance of Point to Point Transfers in the IBM 9076 Parallel Computer. In *Proceedings of Supercomputing 94*, pages 620–629, Washington, D.C., 1994.

[5] Lucas Womack, Ronald Mraz, and Abraham Mendlson. A Study of Virtual Memory MTU Reassembly within the PowerPC Architecture. In *Proc. of the 5th Int. Sym. on Modelling, Analysis and Simulation of Computer and Telecommunications Systems, MASCOTS*, pages 81–90, Haifa, Israel, 1997.