Run-Time Monitoring of Concurrent Programs on the Cedar Multiprocessor.*

Sanjay Sharma Allen D. Malony Michael W. Berry
Priyamvada Sinvhal-Sharma

Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign, Urbana, Illinois 61801

Abstract

The ability to understand the behavior of concurrent programs depends greatly on the facilities available to monitor execution and present the results to the user. Beyond the basic profiling tools that collect data for post-mortem viewing, explorative use of multiprocessor computer systems demands a dynamic monitoring environment capable of providing run-time access to program performance. A prototype of such an environment has been built for the Cedar multiprocessor. This paper describes the design of the infrastructure enabling run-time monitoring of parallel Cedar applications and the communication of execution data among physically distributed machines. An application for matrix visualization is used to highlight important aspects of the system.

1 Introduction

The standard scenario for investigating the behavior of a parallel application is to run the program with profiling enabled, look at the summary statistics after the program completes, modify the program code, and then repeat the cycle. The problems with this approach are two-fold. First, evaluating the behavior of parallel applications running on multiprocessor computer systems generally requires more comprehensive support for monitoring program execution. Simple profiling approaches that condense execution dynamics into summary statistics hide run-time behavior. Tools based on program event tracing are needed to capture data about time-dependent operation.

The second problem deals with the separation of execution from analysis. That is, in the standard scenario, any execution data gathered at run-time cannot be analyzed until after the program terminates. Having the ability to interact with the data as it is being generated makes possible several useful forms of program investigation. For instance, the user might be interested only in certain execution parameters at specific points during the program. This data could be filtered on-the-fly and passed to the user at run-time. Communications support, in addition to the monitoring support, would allow run-time execution data to be passed within a distributed system enabling remote viewing of the data as it is being produced. A feedback control path could also be provided to give the user a level of interactive query capabilities for accessing execution information or guidance control for selectively altering program operation.

We have developed a prototype run-time performance monitoring environment for the Cedar multiprocessor [1, 2, 3] that addresses the two problems discussed above. It has extended monitoring capabilities based on software event tracing with instrumentation support at the user, language, and OS levels. The techniques also include mechanisms for the dynamic off-loading and communication of execution data to remote processes for analysis and display.

In this paper, we describe how the Cedar per-

*This work was supported by the National Science Foundation under Grant No NSF MIP-88-07775, NSF CCR-8717942, NSF-ASC-84-04556, NASA Ames Research Center under Grant No. NASA NCC 2-559 ,the Air Force Office of Scientific under Grant No. AFOSR-90-0044,and the U.S. Department of Energy under Grant No. US DOE-DE-FG02-85ER25001.

CH2916-5/90/0000/0784/$01.00 © IEEE

formance monitoring infrastructure is designed to achieve run-time monitoring and distributed communication of parallel program execution data. The concepts discussed are general in nature and would apply equally well to other shared memory multiprocessor computer systems. Section 3 briefly describes the underlying tracing procedures used in Cedar to gather execution data. Section 4 explains the procedures by which traces are dynamically off-loaded from the Cedar system and sent to remote processes. In Section 5, we apply the tools to capture and visualize matrix data from an application at run-time. Finally, in Section 6, we present results regarding the performance of the run-time monitoring system in terms of execution data bandwidth and program perturbation.

2 Related Research

Tracing software events has been an important technique for monitoring programs with most of the work has been focused on distributed systems [4, 5, 6]. However, several different implementation approaches exist. As an example, the monitoring facility in IDD [4] forces the program to run as a child process of a monitoring process. The event-driven monitoring in [5] supports parallel program monitoring but does not attempt to minimize overhead or intrusion that may be caused by disk I/O, nor does the approach support real-time monitoring capabilities. Monitoring techniques developed for multiprocessor systems [7, 8, 9] includes approaches for real-time concurrent checkpointing. The technique reported in [7] freezes the execution of threads to record the state of the computation before the threads can resume the execution.

Our approach resembles that of hybrid monitoring reported in [6], which includes hardware support for real-time performance monitoring. The difference lies in the advantage of using solely software monitoring. Inherently, software monitoring is more portable than hardware monitoring and has lower development costs. The hybrid monitor developed in [6] uses a central hardware station to interpret the logs, whereas our monitoring infrastructure is independent of any special hardware , and is portable. Our technique is transparent to the user and is capable of providing concurrent program execution details at several levels. Furthermore, a user can take advantage of the standard tools provided as a part of the run-time monitoring infrastructure or can develop his/her own tools to analyze the program behavior.

3 Event Tracing on Cedar

A parallel program running on the Cedar multiprocessor system uses the multitasking capabilities of the XYLEM operating system [10] to partition itself into individual tasks. Each task can take advantage of hardware concurrency support on a cluster, an Alliant FX/8, to execute code in parallel on as many as eight processors.

Programs executing on the Cedar machine are monitored by tracing software events. In addition to the events defined in Xylem and Cedar Fortran [11], the programmer can define events at the user level. All events will be written to trace buffers in the format shown below:

Event Identifier
Concurrency Status
Data Size
Time Stamp
Optional Data

A separate trace buffer is allocated for each execution thread of each task in the program. The user can either allocate large trace buffers at task initialization *(static buffer allocation)* or have trace buffers dynamically allocated and linked during execution *(dynamic buffer allocation)*. In the latter case, trace buffers for all the tasks are retrieved from a shared pool. The tracing facility also allows the user to select between the placement of trace buffers in a task's private memory (cluster memory) or in the shared global memory of Cedar.

The control information for each tracing task is stored in a separate structure called the *task control block*. Task control blocks are maintained by the run-time trace buffer management software. Whenever a tasks is created and initiates event tracing , a task control block is added to a global task control list. Figure 1 shows the organization of the task control blocks and dynamic task buffers.

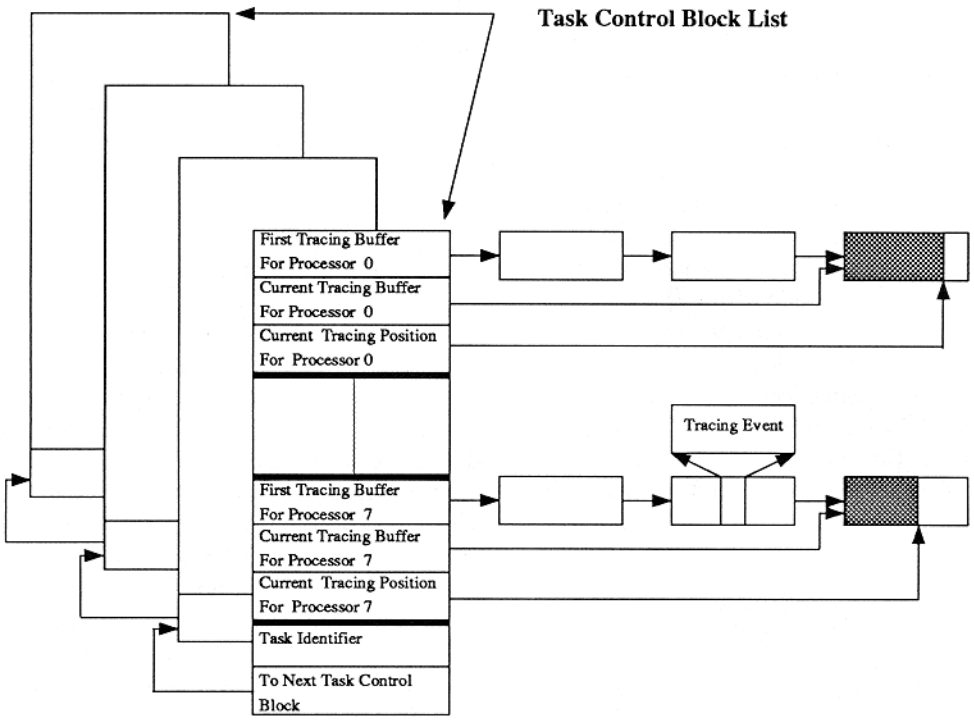

Figure 1: Task Control Block Structure

4 Run-Time Monitoring Support

The run-time monitoring and communications support is provided by a parasitic task called the *snooper task*. The snooper task serves two purposes:

1. to off-load trace data from Cedar, and
2. to support network communication.

The snooper task is scheduled by Xylem like any other user task, however, it is bound to a cluster and cannot migrate. The main reason for wanting the snooper task permanently bound to a particular cluster is to have direct access to disks and network interfaces resident on that cluster.

When a user initiates the snooper task, he can select a file as the destination for the trace data or a network communication to a remote host. Additionally, several parameters can be set to control the snooper task's operation; see below.

4.1 Dynamic off-loading

As implied above, there can be four different choices for how event tracing information is recorded:

1. static tracing and buffers reside in shared global memory,
2. static tracing and the buffers reside in private cluster memory,
3. dynamic tracing and buffers reside in shared global memory, and
4. dynamic tracing and the buffers reside in private cluster memory.

However, in order to provide dynamic off-loading of trace data, all trace buffers must reside in an area of memory accessible by the snooper task. The shared global memory is the only memory in the Cedar system where the snooper task can access all other task trace buffers. Thus, options 1 and 3 can only be used during dynamic off-loading of trace data.

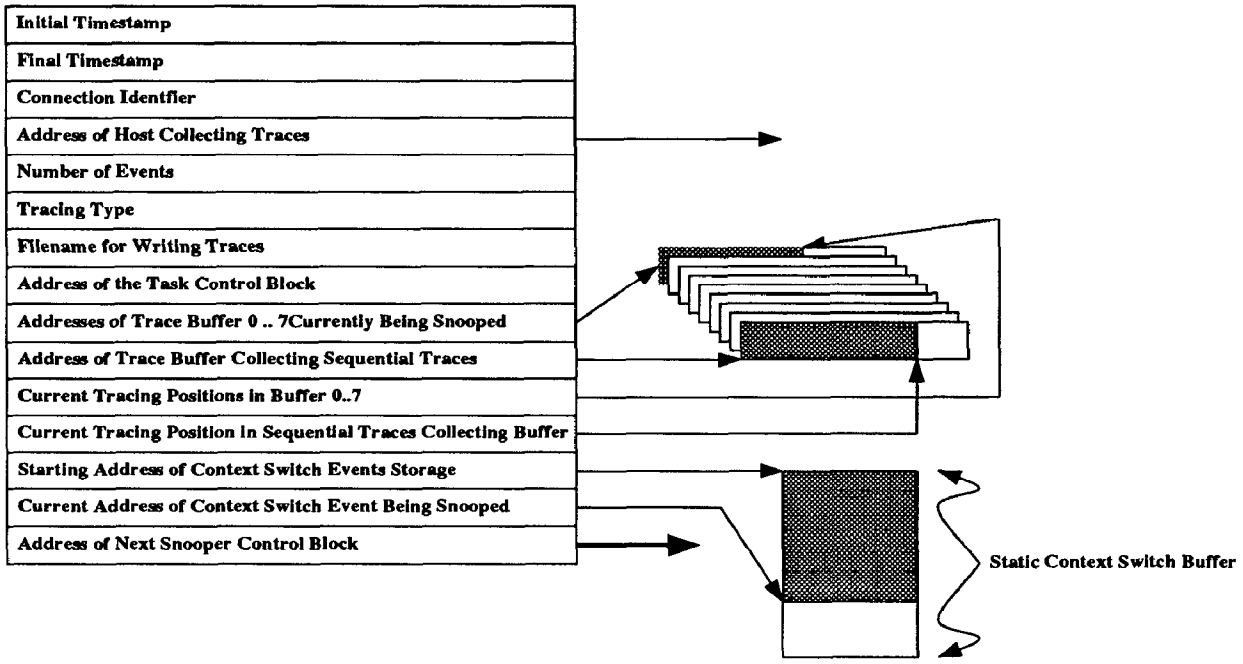

Figure 2: Snooping Control Block Structure

The snooper task continually reads the global task control block list during dynamic off-loading to update its own "shadow" control block list; see Figure 2. The snooper assigns a snooping control block for each task being snooped and copies most of the information from the task control block. It also allocates storage for maintaining run-time parameters for each task, such as initial timestamp, final timestamp, and number of events collected.

The snooper operates by periodically reading task event traces and writing all events occurring within a time window to a file or over the network. The following actions are performed in order by the snooper:

1. take a timestamp to establish the end of the current time window,

2. take a snapshot of the current global control list and update the control block list,

3. sort and write all the events for the currently active tasks occurring before the timestamp (i.e. all events within the current time window),

4. sleep for a user-specified period of time, and

5. repeat until no more events remain and tracing is complete.

In addition, if dynamic trace buffer allocation is selected, the snooper is also responsible for releasing task trace buffers back to the shared pool.

The snooper task must continually update its control block list because new tracing tasks can be created at any time. During a time window, the snooper only looks for the events from currently active tracing tasks registered in its control list. Because the timestamp is taken before the snooper's control list is updated, the snooper is guaranteed to see all events that occur within the time window even though new tracing tasks might have been created after the snooper's timestamp.

The timestamp is taken from a high-resolution, real-time clock. Each of Cedar's clusters has a 10 microsecond real-time clock which measures the elapsed time since last boot. Considering that all the clusters might not have booted simultaneously, two simultaneous events occurring concurrently on separate clusters might have different timestamps. The snooper normalizes the timestamp by maintaining boot time differences between the cluster to correctly handle these timing inconsistencies. These time differences are taken

into account when the snooper calculates its current time window.

The snooper synchronizes the event collecting operation with the event tracing performed by a task. An event is timestamped only when the event has been completely written to the trace buffer. This avoids the conflict that may arise from partially stored events. Only those events with a timestamp within the current snooping window will be accessed.

The snooper task sorts events collected during a time window on per task basis. The traces from the different task execution threads are merged into a single time-ordered event stream. This allows separate task trace files to be produced or, with networked communications, separate remote destinations to be supplied separate task trace streams.

A *sleeping factor* parameter can be set by the user to control how often the snooper checks for new tasks and new events. Because the snooper task is competing for resources with other program tasks, the user can reduce snooper overhead by setting a long sleeping interval. However, the user must assess the tradeoff between snooper overhead and the degree of real-time access to the trace data.

4.2 Network communication support

The network communication support allows the execution behavior of parallel programs executing on Cedar to be analyzed in real-time on another machine. This alternative is selected instead of file I/O by providing the snooper with a hostname address at the time it is initialized. Once communication is established with the specified host, all events handled by the snooper will be sent to that host.

The network support is built on top of the 4.2 BSD Unix network primitives [12]. As shown in the Figure 3, the snooper first establishes communications with a special daemon process (referred here as the *tracer*) on the destination machine and follows a client-server protocol. It is important to note that a separate snooper task and remote communications channel can be created for each tracing application. Thus, multiple applications can be tracing simultaneously with each sending execution data to remote processes.

Once the tracer is established, it creates a child process that will be the destination for the event data streams. The child process typically performs real-time event filtering and interpretation, but the child process can be any application the user chooses. The only requirement is that it support the protocol for the snooper-to-child data communications. The format of the data packet passed between the snooper task and the child process is shown below:

Total Packet Size
Task Identifier
Number Of Tracing Events
Events
Checksum

The snooper constructs a packet by specifying the packet size, the task identifier of the task from which the data was generated, the number of software events in the packet, and a checksum. The snooper off-loads all the merged events occurring within the time window before collecting new events. Primitive control commands from the child process back to the snooper are supported to specify, for instance, which trace streams are to be sent over the channel.

5 User Instrumentation

A library of tracing and monitoring control routines are supported. The *trace_event(eid)* routine records the event, *eid*, in a trace buffer determined by the calling task identifier where the event occurred. The *trace_data(eid,data,size)* routine stores *size* bytes of data in addition to the information stored by the *trace_event()* routine. The snooper task is invoked by the *init_snooper(host,sfactor,type)* routine where *host* is the destination of event streams for network communication, *sfactor* controls how long snooper task is sleeping before it checks for new tasks and events, and *type* connects the event streams to one of various deamon processes on the destination host.

6 A Matrix Visualization Example

The design of efficient yet robust numerical algorithms for the complex architectures of super-

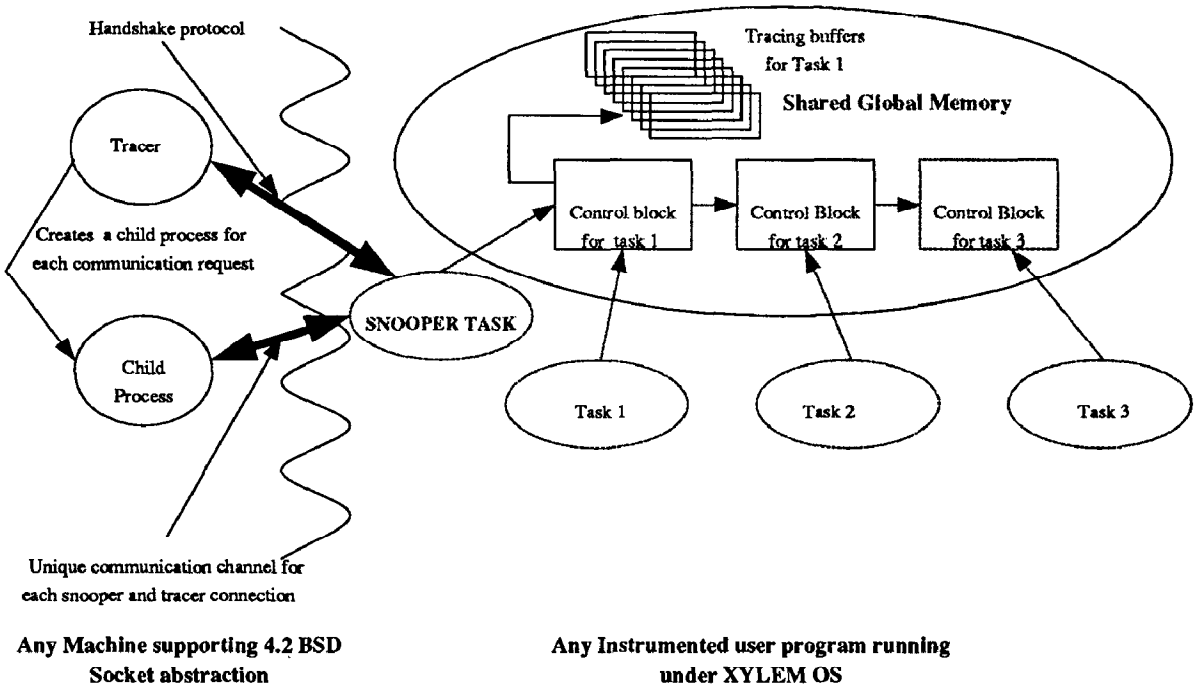

Figure 3: Snooper Communication Architecture

computers can be greatly expedited through the use of color graphics. By visualizing the run-time behavior of an algorithm through its most fundamental data structures (e.g. matrices), a numerical analyst or scientific programmer can immediately detect unknown phenomena or errors in the algorithm's logic. For example, the detection of separable dataflows or independent (parallel) subproblems is extremely desirable in order for algorithms to exploit multiprocessor, multicluster architectures such as Cedar. As discussed in [13], the matrix visualization package, *MatVu*, can be easily used to classify the convergence patterns of classical iterative eigenvalue algorithms (Jacobi) by assigning a logarithmically scaled color table to the magnitudes of the off-diagonal elements in each sweep of the particular (Jacobi) algorithm. A significant discovery from this effort has been the eventual convergence to a preliminary block diagonal form for matrices having clustered eigenvalues.

Figure 4 is a black-and-white illustration of the numerical decoupling of a larger eigenvalue problem having four (equal-sized) clusters (center window) into distinct (parallel) subproblems (the four corner windows) when an appropriate context-switching criterion is satisfied. The run-time monitoring support was used to allow a user to instantly observe the separate convergence associated with these parallel subproblems. The user interface to instrument the program is simple; see the Figure 5. The matrix-related computations are stored in the buffer by *trace_data()* calls. The snooper task is invoked as described early in the section.

As shown is Figure 6, a child process is created to collect matrix visualization data. The child process creates a backing store for the data and starts the *MatVu* application with a pointer to this backing store. The *Matvu* process synchronizes with the child process by a communications pipe. When the child process receives the encapsulated matrix data, it filters it into the format required by *MatVu* and writes them to backing store. A message is posted in the pipe when sufficient data have been received. *MatVu* reads the data from the backing store and interprets it to visualize the current state of the matrix compo-

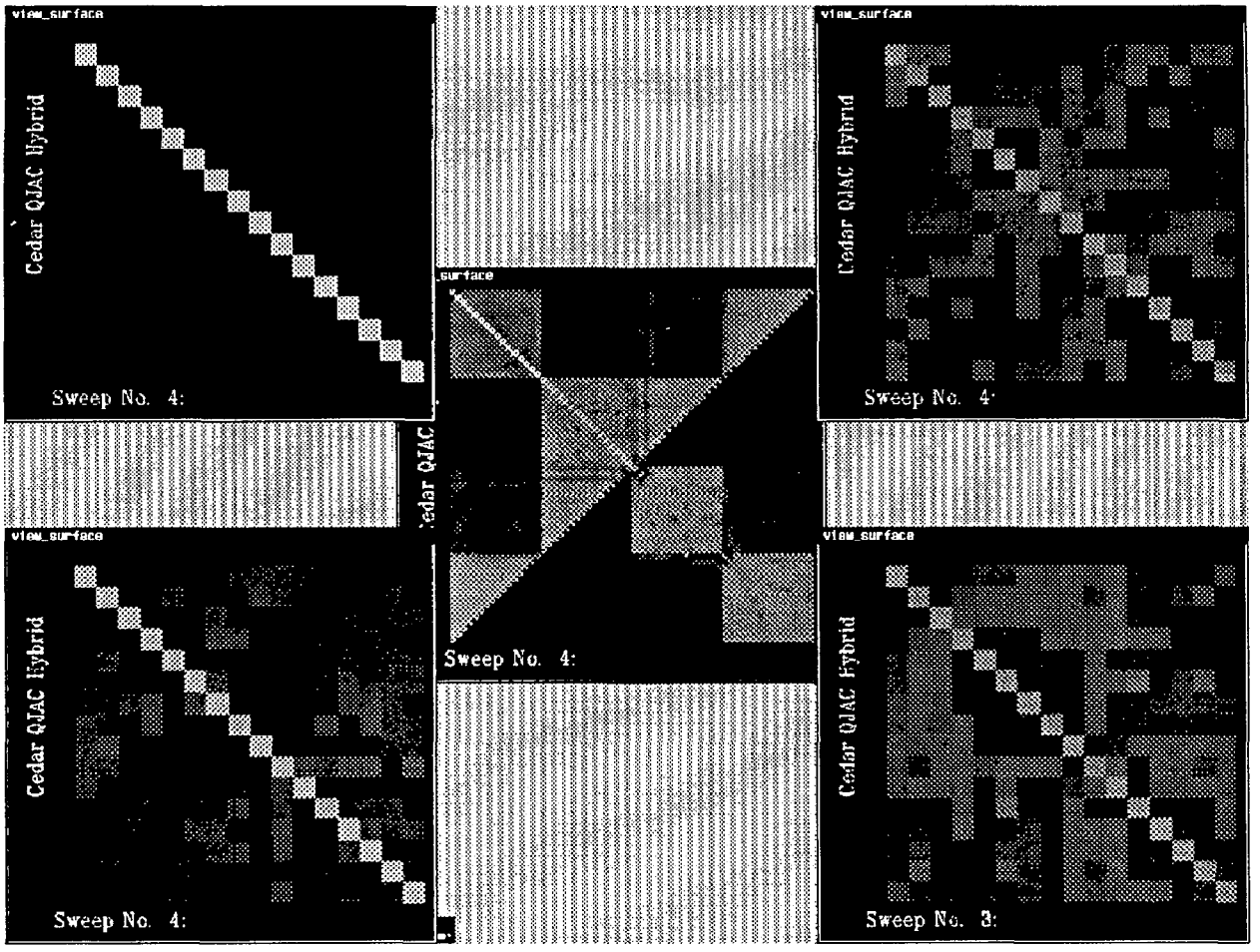

Figure 4: Visualization of a numerical decoupling of one Cedar cluster task into four smaller independent (parallel) Cedar cluster tasks via the snooper task and MatVu.

nents. In Figure 4, each of the four off-centered windows displays matrix data generated by one of the four Cedar cluster tasks which are executing in parallel after the decoupling is made.

By interfacing MatVu with run-time monitoring, we can interactively study the robustness of algorithms across several problem domains (e.g., various matrix orders and spectra). Also, high-level *debugging* capabilities for interpreting run-time errors not only in source code but also in an algorithm's logic are possible. The usual playback of massive trace outputs can be avoided by easily identifying the time and location of the error as revealed in the matrices displayed via MatVu and communicated by the snooper task. Another attractive feature is the ability to run an application on the target supercomputer (Cedar in this case) and observe its behavior in pseudo real-time on a local desk-top workstation.

7 Timing Results

Because the run-time monitoring and remote communications operate while the program executes, it is important to determine not only the monitor's performance, but also the perturbation on the program's behavior. Experiments were performed to determine the following:

- program perturbation introduced by run-time monitoring,
- snooper communications bandwidth as a function of packet size, and

```
C
C     INFO ARRAY:
C     (1) ITERATION NUMBER
C     (2) ORDER OF ITERATION MATRIX
C     (3) STARTING ROW INDEX FOR TRACE_DATA
C     (4) STARTING COL INDEX FOR TRACE_DATA
      MATVU1=1
      MATVU2=2
      ISIZE1=8*N*N
      ISIZE2=4*4
      INFO(2)=N
      INFO(3)=1
      INFO(4)=1
C
      CALL TRACE_DATA(MATVU2,INFO(1),ISIZE2)
      CALL TRACE_DATA(MATVU1,MATRIX(INFO(3),INFO(4)),ISIZE1)
C
```

Figure 5: Example of Fortran source code instrumentation for capturing iteration matrices which will be rendered by MatVu.

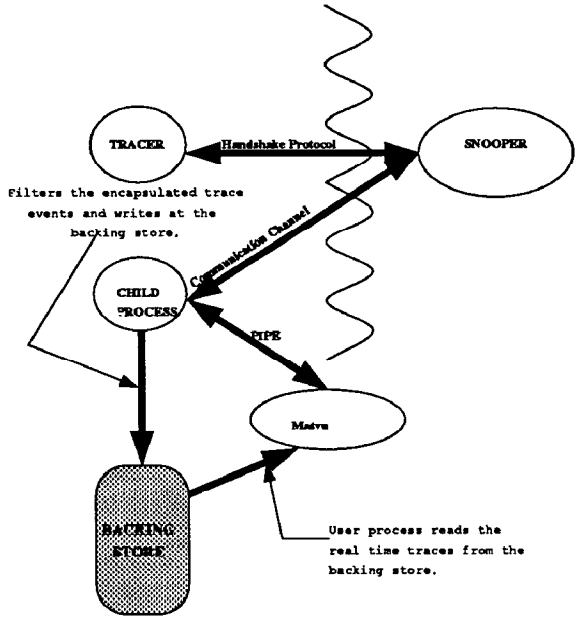

Figure 6: Run-Time monitoring integration with MatVu

- snooper behavior as a function of sleeping factor.

The program perturbation was measured on a simple problem containing nested DO-loops. The number of iterations was varied in the inner loop, and execution times for the inner loop were measured to determine the perturbation effects of calling trace_data() and off-loading trace data by the snooper task. The outer loop count was kept fixed at 10000 iterations. In order to measure the maximum intrusion by snooper task, the problem was divided equally on the four Cedar clusters. The results are shown in Figure 7.

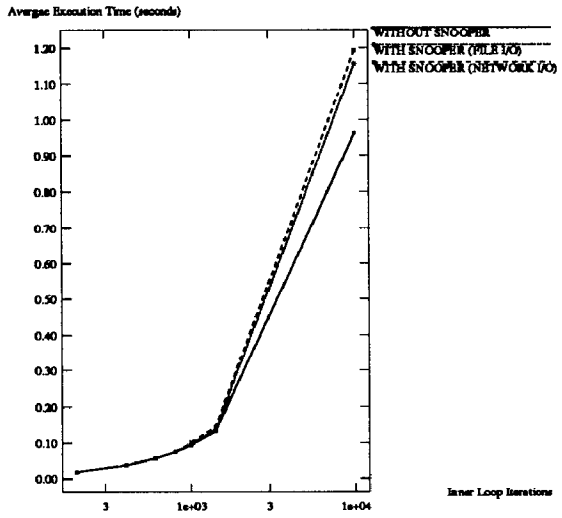

Figure 7: Variation in execution time by introducing the snooper task and doing remote I/O

Several interesting points are observed. The snooper intrusion increases as the problem size increases. This is mostly due to increased number of events. The increase in execution time was greater when doing network I/O as opposed to file I/O because of the additional protocol over-

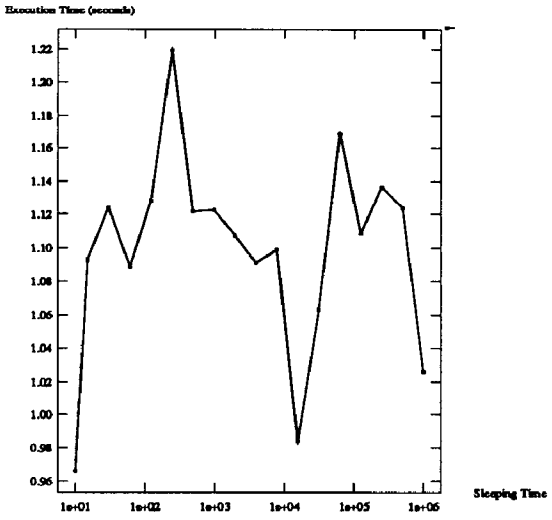

Figure 8: Program execution perturbation

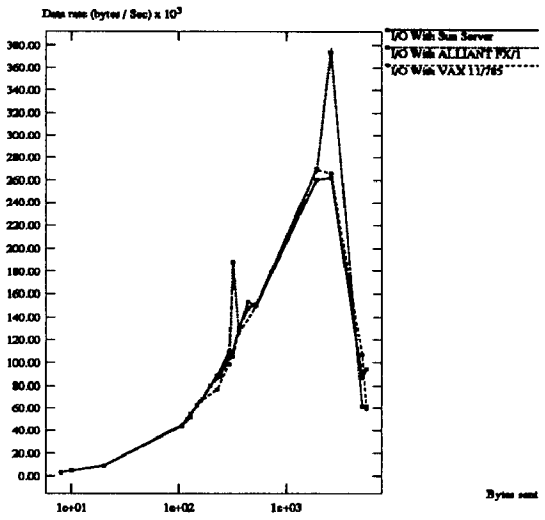

Figure 9: Bandwidth Variations

head and the reduced bandwidth; a 10 Mbits/sec Ethernet is used as the network. Execution times differed by as much as 20% with the snooper task writing task traces to files but up to 24% with network communications.

The file I/O experiments were performed in dedicated mode. The network experiments, on the other hand, were not, and could have been impacted by other users on the network. This will be the case in general, although we performed the experiments during a time when the network would have been lightly loaded. Even in dedicated mode, the asynchronous nature of scheduling program tasks under Xylem can have a non-uniform influence when different snooper sleeping values are selected.

Figure 8 shows the execution time for 10000 iteration of the do loop (no remote I/O) relative to sleeping interval. The times shown are the best observed of 5 runs. Although the minimum time is within 18% of the maximum time, the ratio does not follow a uniform pattern. This suggests that different sleeping intervals might have different effects on execution behavior.

We also measured trace data bandwidth for different combinations of networked machines to determine what differences there might be on the performance of run-time monitoring; see Figure 9. The data rate improves with increasing packet size but peaks when the packet size reaches 2600 bytes. Beyond this point, the bandwidth falls off dramatically because of limits in lower-level I/O buffering by underlying socket implementation. Notice that the packet size depends on the number of events collected in a snooper time interval. We also see differences between server combinations, but only in the region of peak bandwidth. Thus, to achieve the highest network data rates possible during run-time monitoring, we not only have to be cognizant of the packet size and the limitations of the server, but also the the variables affecting execution data production, namely:

- the number of tasks created,
- the number of tracing calls performed, and
- the snooper task sleeping interval.

8 Conclusion

Performance evaluation environments for parallel computer systems in the future should include some form of dynamic run-time monitoring of concurrent program execution. There are

significant advantages to be gained from having the ability to view program behavior interactively during program execution. Many performance and debugging problems can be determined by filtering execution data in real-time for anomalous conditions, avoiding storing potentially large execution histories for port-mortem review. There is also the interesting possibility of interacting with a program's execution either by selecting different parameters for monitoring or through a feedback path whereby certain execution-control variables can be adjusted. The performance environment built for the Cedar machine is a prototype of a dynamic run-time monitoring system. Many of the design ideas implemented would apply equally well to other shared memory machines. Currently, we are extending the environment to support interactive run-time visualization of data structures used in scientific programs.

References

[1] D. Gajski, D. L. Kuck, D. Lawrie, and A. Sameh, *Cedar - A Large Scale Multiprocessor*, Proceedings 1983 International Conference on Parallel Processing, Belaire, MI, 1983.

[2] D. J. Kuck, A. H. Sameh, *A Supercomputing Performance Evaluation Plan*, Proceedings 1987 Supercomputing Conference, Greece, June, 1987.

[3] K. Gallivan, W. Jalby, A. Malony and P.-C. Yew, *Performance Analysis on the Cedar System*, CSRD Report No. 680, University of Illinois at Urbana-Champaign, June, 1988.

[4] P.K. Harter, D.M.Heimbigner and R.King, *IDD: An Interactive Distributed Debugger*, in Proc. of Distributed Computing Systems, May, 1985, pp. 498 - 506.

[5] T. J. LeBlanc and A D. Robbins, *An Event-Driven Monitoring of Distributed Programs*, in Proc. of Distributed Computing Systems, May, 1985, pp. 515 - 522.

[6] D. Wybranietz and D. Haban, *A Hybrid Monitor for Behavior and Performance Analysis of Distributed Systems*, IEEE Transaction of Software Engineering, Vol. 16, No.2 ,pp. 197 - 211.

[7] Kai Lai, J.F.Naughton and James S. Plank, *Real-Time, Concurrent Checkpoint for Parallel Programs*, in Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming, March 14-16, 1990, pp. 79-88.

[8] R. J. Fowler, T. J. LeBlanc and J.M. Melbr-Crummey, *An Integrated Approach to Parallel Program Debugging and Performance Analysis on Large Scale Multiprocessors*, Proceedings of the workshop on Parallel and Distributed Debugging, ACM SIGPLAN/SIGOPS, 163-173, 1988.

[9] T. Lehr, Z. Segal, D. F. Vrsalovic, E. Caplan, A. Chung, and C. E. Fineman, *Visualizing Performance Debugging*, Computer, October 1989, pp. 38-52.

[10] P. Emrath, *An Operating System for the Cedar Multiprocessor*, IEEE Software, Vol. 2, No. 4, 1985, pp. 30–37.

[11] M. D. Guzzi, *Cedar Fortran Programming Handbook*, CSRD Report No. 601, Univ. of Illinois at Urbana-Champaign, Urbana, 1987.

[12] University of California. *Unix User's Manual, Reference Guide-4.2 Berkeley Software Distribution*, Computer Science Division, University of California, Berkeley, California 1984.

[13] A. Tuchman and M. Berry, *Matrix Visualization in the Design of Numerical Algorithms*, CSRD Report No. 826, Univ. of Illinois at Urbana-Champaign, Urbana, 1989, to appear in ORSA Journal of Computing 2:1(1990).

[14] Allen D. Malony, *Program Tracing in Cedar*, CSRD Report No. 660, University of Illinois at Urbana-Champaign, April, 1987.