dproc - Extensible Run-Time Resource Monitoring for Cluster Applications

Jasmina Jancic, Christian Poellabauer, Karsten Schwan, Matthew Wolf, and Neil Bright

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
{jasmina, chris, schwan, mwolf, ncb}@cc.gatech.edu

Abstract. Large-scale clusters require run-time monitoring of their resources to determine the appropriate allocations of resources to applications and to ensure the applications requirements for performance or quality of service (QoS). Such applications share I/O devices, access remote sensors and large-scale remote data contained in digital libraries, support scientific collaboration by remote visualization of their data, and they interact with other computations via the Grid[1, 2].
We describe the *dproc* (distributed /proc) kernel-level mechanisms and abstractions, which provide the building blocks for implementation of efficient, cluster-wide, and application-specific performance monitoring. Such monitoring functionality may be constructed at any time, both before and during application invocation, and can include dynamic run-time extensions. This paper (i) presents dproc's implementation in a Linux-based cluster of SMP-machines, (ii) evaluates its utility by construction of sample monitoring functionality, and (iii) shows its effect on the performance of clustered applications.

1 Introduction

Motivation. Run-time monitoring of large-scale cluster machines is critical to the successful operation of cluster applications. This is because even a single high performance application running on a cluster typically exhibits highly dynamic computational behavior. Moreover, most applications do not run in isolation: they conduct I/O, require real-time data from remote sensors[3], access large-scale remote data contained in digital libraries or share files across the computational grid, support scientific collaboration by remote visualization of their data[21, 22], and interact with other computations via the Grid [1][1, 2]. Unless run-time monitoring is used to determine the appropriate and dynamic allocation of cluster resources to applications[7, 11], high performance is unlikely to be attained.

Run-time monitoring mechanisms are required to dynamically diagnose the performance of cluster programs. The monitoring tools commonly available to

[1] http://www.globus.org/

cluster programmers, however, are not only used for adjusting programs[25, 12] at runtime, but they are also used for diagnosing performance problems at the time of program implementation, for program profiling[23], and even for debugging them[18]. As a result, developers routinely impose a wide range of requirements on such tools, including:

Selective Monitoring of Multiple resources. For cluster machines, at minimum, monitoring must capture usage and availability of both CPUs and network links. Often, additional information is required, as evident from the rich performance data routinely available from current monitoring tools for high performance machines[23, 24]. For large cluster programs, resulting overheads make it infeasible to capture all such data about all nodes at all times. Thus, monitoring must be performed selectively, applied dynamically to precisely the resources and program components under investigation.

Distinguishing multiple resource users. To understand the performance of specific application programs, it must be possible to distinguish the resources used by an application from those used by other applications, the operating system or the middleware.

Variable granularity. Fine-grain monitoring data is needed for certain optimizations of applications, such as recognizing the precise arrival times of processes at shared barriers, understanding the actual overlap in communication and computation attained by a code[8], or diagnosing the degree of simultaneity in communications and thus, the potential network loads being imposed.Therefore, it should be possible to conduct monitoring at variable frequencies and rates, thereby altering the precision vs. perturbation induced by monitoring.

Flexible and dynamic analysis. It is well-known that monitoring data should be condensed and filtered as 'close' as possible to its points of capture, to reduce monitoring overheads and perturbation[23, 7]. However, the actual analyses to be performed typically depend on what monitoring is currently used for, and such analyses vary in their behavior, some causing little perturbation, others requiring substantial trace data before they may be applied. No single built-in set of analysis routines will satisfy all applications. Furthermore,especially for long running applications, it is not viable to install all monitoring support once, then simply use it. Instead, monitoring should be installed at runtime[23], analyses must be changed as needed[7], and its monitoring overheads should be dynamically controlled.

Standard interfaces. Monitoring information may be used repeatedly: for replay, to share it with fellow developers, or to store it until future use. This implies the need for standard interfaces[5] which can then export data for use by graphical tools[13, 18], and for long term storage.

The dproc approach to performance monitoring. This paper describes kernel-level mechanisms and abstractions that are the building blocks for cluster-wide performance monitoring. Their realizations in a Linux-based cluster of SMP machines are evaluated by construction of dynamically extensible and changeable monitoring functionality. Cluster resources monitored include both node and network attributes, including CPU loads, memory and swap usage, achieved communication bandwidth, loss rate and message round-trip times. The API the tool presents to programmers is an extension of the standard /proc performance interface offered by Linux systems, hence motivating the use of the term dproc for our facilities.

Dproc offers the following functionality:

Selective monitoring via kernel-level publish/subscribe channels. The basic operating system construct offered by dproc is that of *monitoring channel* (monchannel). A single monchannel can capture monitoring information from any number of sources, and it can distribute it to any number of interested parties (sinks). Sources or sinks may reside at user- or at kernel-level. In this fashion, a monchannel can capture monitoring data from multiple resources, and the results of such monitoring can be distributed to whomever requires such data (e.g., performance displays, data storage engines).

Standard API. Applications need not explicitly handle monchannels. An application accesses dproc entries, which are physically represented by underlying monchannels, Through the standard /proc pseudo-file system interface. Kernel modules perform monitoring by publishing data described as *monitoring attributes* (monattributes) on channels and by listening for attribute updates. Applications simply access the dproc entries that correspond to such attributes. The hierarchically structured dproc API is organized such that it is easy to access (1) per node attributes, (2) per process/thread attributes, and (3) per application attributes.

Differential control. Dproc offers simple ways of dynamically varying certain parameters of monitoring actions, such as monitoring rates or frequencies. Specifically, with each monchannel is associated an implicitly defined *control channel* via which control commands are propagated from monchannel sinks to sources. The dproc interface gives applications access to selected control commands via *control attributes* also maintained with dproc entries.

Flexible analysis and filtering. In order to permit monitoring data to be filtered and analyzed when captured at its sources, monchannel creators can define analysis functions, termed *monhandlers.* These handlers are applied to monitoring data at the sources of channels, thus enabling data filtering and condensation. A monhandler is executed every time an information item is submitted to the channel. A simple but nontrivial example of a monhandler is one that provides window-based running averages rather than raw data. To deploy monhandler functions at kernel level, dproc offers a simplified way of linking an appropriate handler function into the local kernel. For deployment across machines, the remote dynamic code generation facility provided by KECho is used.

Runtime configuration. Monchannels, handlers, and control attributes may be created, changed, and deleted at any time during the operation of dproc. In this fashion, new monitoring functionality can be added on the fly, and existing functionality can be altered or removed.

Dproc and XML interfaces. In addition to the /proc interface, dproc externalizes monitoring data as XML records using schemas that describe the low-level, efficient binary formats in which monitoring information is carried[4]. In this fashion, dproc data is accessible efficiently via its system call interface and in an interoperable fashion to the wide variety of XML-capable display tools and storage facilities now being developed[5].

Related work. Cluster monitoring tools typically rely on the use of daemon processes installed on participating cluster nodes[15, 14]. As a result, they cannot capture data with the overheads and granularity offered by dproc. Similarly, when monitoring data is collected and maintained by single (or multiple, hierarchically arranged[7]) monitoring 'master' processes, data may be captured and analyzed efficiently and with high throughput, but such monitoring structures suffer from high latency in data access. This is important when monitoring is used for online program tuning or steering[7, 25].

Dproc could benefit from additional performance information captured at the network or switch levels[6, 28]. At this time, we are implementing network monitoring by inspection of kernel-resident protocol stacks using the kernel-resident libpcap portion of the well-known tcpdump facility.

Higher level services that interpret or analyze monitoring data[16, 24] are not the subject of this research, but would be useful when using monitoring data for runtime program steering[25] or to help programmers tune their cluster applications[24].

Dproc uses the open source nature of the Linux kernel and its ability to dynamically link new modules into the kernel, to implement efficient kernel-kernel communications using KECho, to instrument the kernel, and to execute monitoring functions that update dproc in the kernel. For other platforms, instrumentation might be performed with runtime binary editing, as described in [23]. Without KECho, a user-level implementation of dproc can still offer efficient access to monitoring information via the dproc API, but cannot collect monitoring data with the granularity shown in this paper. Martin et al.[10] demonstrated reduced overheads by embedding monitoring functions into network co-processors rather than operating system kernels. We are experimenting with that approach in a related project[20].

Overview. The remainder of this paper first outlines the software architecture, API, and implementation of dproc. Section 3 evaluates dproc with microbenchmarks and by applying it to improve application performance. Conclusions and future work are outlined in Section 4.

2 Dproc Architecture and Implementation

Overview. Procfs is a standard component of a Linux file system structure, which offers performance characteristics of the local system. For instance, /proc/meminfo provides statistics about memory and swap usage, buffer and cache sizes and utilization, etc. Unlike standard Unix file systems, viewing files in procfs essentially executes a piece of code that collects this information dynamically, on-demand. Such information is extracted from the kernel data structures, and is updated by the kernel.

Dproc is a distributed extension of /proc that provides hierarchically organized, application-specific views of monitoring information about both local and remote cluster nodes. For instance, viewing /dproc/cluster/node1/meminfo will provide information about memory statitistics on node1. Thus, through calls to the local dproc API, an application can view the current values of monitoring attributes about remote nodes. For each such attribute, an application can also specify the attribute's update rate and ranges of values of interest, thereby resulting in fine grain control over the performance vs. overheads of monitoring experienced by applications.

Monitoring attributes are updated via kernel-level monitoring channels that 'push' update events from the sources being monitored (i.e., certain cluster nodes) to their sinks (i.e., other cluster nodes), much like it is done in the object-based model of monitoring for distributed systems described in our earlier research [9]. By providing a separate underlying 'monchannel' for each information item captured and distributed, dproc permits its distribution to be performed at a unique rate and/or its filtering or analysis to be performed in a unique manner.

Prototype. Dproc is not a complete monitoring system. Instead, it provides basic building blocks for constructing customized monitoring functionality for target systems and applications. The dproc prototype used in this paper employs a predefined set of monchannels across cluster nodes. If interested in certain monitoring information, a node subscribes to a monchannel as both a source and a sink, and is thereby able to provide information of this type and also receive it. We have implemented two approaches to monitoring: user- and kernel-level. Both approaches use monchannels to distribute monitoring information. The channels are implemented in user and kernel space, respectively. Dproc reads and writes in user-level are performed through standard read() and write() system calls. In the kernel, dproc entries are accessed directly through a special interface. Two facilities of dproc's monchannels contribute to the performance and scalability of monitoring for large-scale cluster machines:

Control attributes appearing in the dproc API permit applications to specify explicit monitoring behaviors, like desired attribute update rates or the ranges of values in which they are interested. Control attributes are implemented by a cluster-wide control channel to which all dproc participants subscribe. The important characteristic of control attributes is that monitoring data is transported from sources to sinks only when the constraints stated by control attributes are met. The intent is to reduce monitoring traffic across cluster nodes. Unless monitoring is turned off, however, once a monchannel has been created, data will always be captured and analyzed at the sources, so that its control constraints can be tested. We have not implemented the sophisticated techniques for dynamic instrumentation needed to entirely eliminate instrumentation overheads, like those described in [23].

Handler functions perform both complex control tasks as well as application- or resource-specific analyses and filtering. A sample handler performing analysis for the kernel-level monitoring of message traffic is one that computes the average message delay experienced over a variable-sized window, for instance. Other examples are functions that filter messages or that compute the actual *utility* applications derive from receiving them[19], by recognizing and computing statistics derived from the application-level header information contained in such messages[17].

Software Architecture. A sample use of dproc monitoring is one in which the current availability of cache space is noted for some set of cluster nodes used by an application program. Information about cache space is captured in the kernels of all 'server' nodes and transported to a single dproc subscriber for this information, located on a 'coordinator' node. This is achieved by using a KECho channel created by the 'coordinator' node, which is registered as a 'consumer'. All server nodes are subscribed as 'producers'. Monchannel handler functions comprise the instrumentation resident in the servers' OS kernels. They are executed at rates determined by the monchannel's control attributes. Each time such a function executes, a monitoring event labeled by node id is submitted to the KECho channel. Upon its receipt, an intra-address space quality event is used to update the 'cache space available' attribute resident in the dproc structure local to the 'coordinator' node.

The dproc API is accessible from any process running on the local machine via a simple system call. A dproc call operates just like a /proc system call, with its resulting overheads corresponding to that of other Linux system calls. The resulting 'pull'-based model of performance monitoring currently implemented by dproc offers high performance, but does not satisfy uses of monitoring for actions like program steering or adaptation, where an application change may be indicated as soon as some condition has become true. Toward this end, we are currently generalizing the call interface offered to dproc to one that supports both the common 'pull' model and a model in which an application can register its interests in certain values and is signaled when these values meet certain conditions. Figure 1 shows dproc architecture.

Extended Functionality The dproc interface defines snapshots of current resource and program states. For long-term storage of such snapshots, for tracing, for postmortem replay, or for online performance visualization via graphical tools, monitoring attributes are also available from files that contain traces of XML data items produced from monchannel attributes. Toward this end, monitoring data is 'externalized' as XML when so desired. It is important to note that monitoring data is never transported using XML. Instead, the xmit conversion tool is used to dynamically transform the binary forms of monitoring events used internally in monchannels to their external XML forms. Details on xmit appear in [27].

We are working on implementing dproc hierarchically, indexed by node id, process id, socket id and in addition, by views explicitly defined by applications. The intent is to provide system-level information in some standard form, while also providing to each application its own views of the targets being monitored. For instance, an application may define a view that identifies some selected

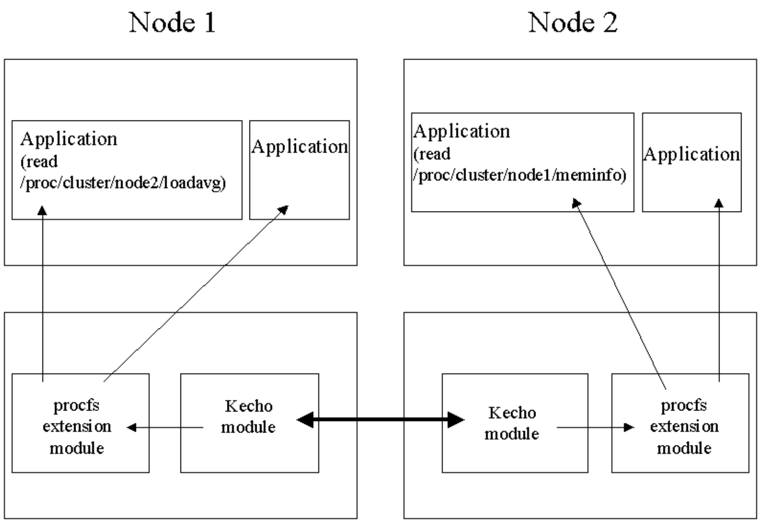

Fig. 1. Dproc Architecture

number of processes on certain cluster machines, or certain network connections it desires to monitor.

Runtime monitoring for large-scale cluster machines must provide ways of reducing the potentially large amounts of monitoring data exchanged between cluster nodes. dproc provides multiple ways of reducing monitoring data, including the placement of application-specific filters of monitoring information in data sources. A simple example of this concept is one where such filters dynamically trade off monitoring granularity vs. perturbation by changing the frequency with which certain monitoring attributes are updated. Another example is an extension of remote nodes to compute a specific composite performance measure needed by a parallel graphics application. By computing measures like these remotely rather than locally, reductions in monitoring traffic are realized. This not only reduces traffic, but allows for dynamic load distribution in the system based on resource availability of different nodes.

3 Experimental Evaluation

All experimental results described in this paper are attained on a group of Pentium II quad-processor machines, interconnected via switched Fast Ethernet, and running the Linux version 2.4.0 kernel. We achieve a fine-grained control of the machine loads by running multiple instances of a CPU intensive application. Microbenchmarks are used to evaluate the basic performance of dproc's mechanisms, such as its KECho kernel-kernel communication channels, its runtime API, and its ability to update an API entry in response to the remote update of its value.

3.1 Microbenchmarks

Since information in cluster dproc is collected from remote nodes, the following issues need to be addressed:

- Total monitoring latency, in terms of the minimum delay experienced for updating a remote dproc API in response to a change of some monitored attribute.

- Ability to provide timely access to monitored information, especially during periods of high system load, as monitoring is crucial in such cases. In addition to low access latency, low deviation in access time is also required. These two conditions ensure that monitoring information can be treated as timely data with reasonable confidence.

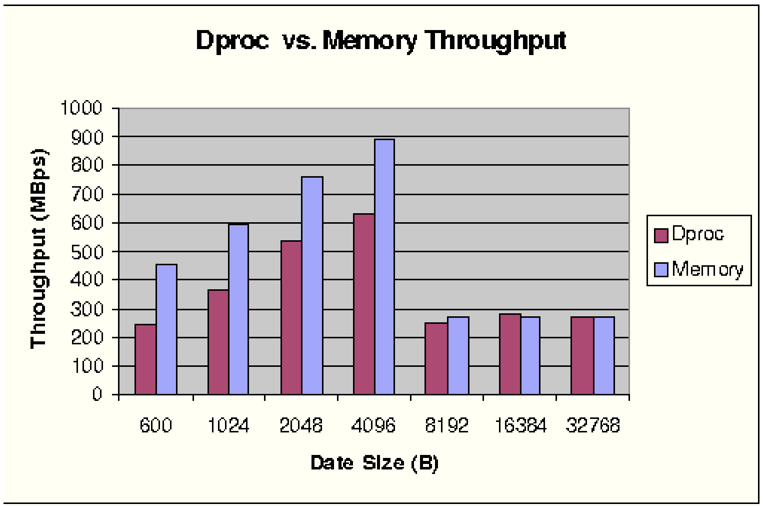

Fig. 2. Dproc Throughput

Figure 2 shows the throughput of dproc, i.e. the time to update a dproc entry as a function of the amount of data. The throughput of dproc is compared to memory throughput. Independent of data size, dproc throughput is comparable to that of memory. Both throughputs decline for data sizes over 4K, as this is the page size, and memory writes now take longer to complete.

Figure 3 compares the total message/event round-trip times in our user and kernel level implementation, respectively. The RTTs are calculated as a function of (symmetric) system load. The results include handler execution times on both the source and the sink. Results from multiple runs are presented to demonstrate the variability of RTT in user and kernel level. Note that we use a logarithmic scale. The results show that a user-level approach to monitoring is especially succeptible to changes in system load, due to the delays experienced on the

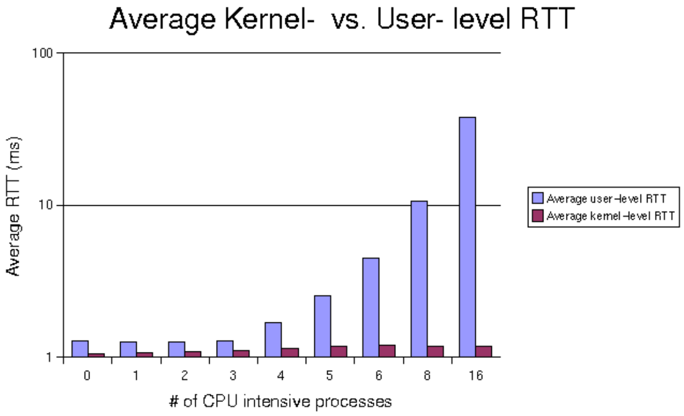

Fig. 3. RTT Variance

run queue as user-level threads compete for CPU cycles. In contrast, kernel-level threads are scheduled more frequently, and with much lower variation. Our experiments also confirm that the RTTs in the user-level approach have a significantly higher standard deviation compared to the kernel-level approach.

Figure 4 shows the delays experienced in accessing monitored information for a daemon-based approach and our user-level implementation as a function of system load. The daemon-based approach is simulated by a server and a client communicating over a socket. This setup approximates a typical monitoring approach which uses a central site to collect, process and distribute monitoring information. We refer to this as a 'pull' based approach, because each interested party has to pull the information from a remote node. In our setup, the client and the server run on the same machine, which is the best-case scenario with the lowest possible network delay. It can be seen that accessing a dproc entry is much more efficient than pulling such information from a remote server via sockets. The access times for dproc are nearly independent on the machine loads, and entirely independent of the network loads.

Preliminary results also show that the perturbation imposed by kernel-level monitoring is negligible for simple monitoring mechanisms used in our experiments (less than 1 percent). We intend to experiment with larger clusters, and quantify the overhead and perturbation imposed by dproc used at both user- and kernel-level.

Microbenchmarks demonstrate that dproc's monitoring overheads and latencies are far better than those experienced by approaches to cluster monitoring that use replicated daemon processes and/or 'pull' monitoring data from remote nodes on demand. In addition to lower latencies, the results demonstrate that monitoring in the kernel provides significantly lower variation in update/access times. This is emphasized when the system is under heavy loads, which implies that user-level daemons experience delays when placed on the run queue. In

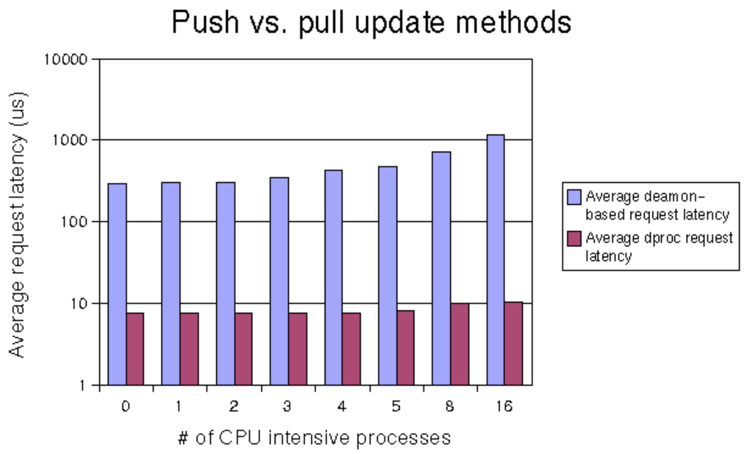

Fig. 4. Read Request Latency

the kernel approach, those delays are reduced, since monitoring functions are executed by kernel threads.

One advantage of the dproc approach to monitoring is that updates of monitoring attributes are performed asynchronously with application programs' inspections of attribute values. In other words, dproc separates the capture and distribution of monitoring attributes from their inspection by applications. The resulting performance improvements attained by dproc are similar to those attained for parallel programs in which communication is overlapped with computation.

4 Conclusions and Future Work

Our approach to monitoring is based on three principles:(1) monitoring should be reliable during periods of heavy system load, as this is the right time to utilize monitored information and make appropriate changes in the system, (2) monitoring should be dynamically extensible and customizable since no single approach to monitoring will satisfy the needs of all applications, and (3) monitoring overhead and perturbation should be minimized and adjustable. To address the first issue, we built dproc as a kernel-level facility, thereby ensuring that reaction time of the monitoring system, and its variation are minimized. The other two issues are addressed by the extended dproc functionality: filtering and dynamic configuration/extension of the system allow for scaling, reducing and controlling perturbation. Experimental results demonstrate that the user level monitoring benefits from the fact that information is available locally on the requesting node, essentially implementing a form of caching of remote data, and consequently reducing access time compared to daemon-based (pull) approaches. Kernel-level monitoring benefits from the richness of information available in the kernel, and from the immediate thread scheduling available there. As a result, it exhibits much lower update/access times and offers greater flexibility.

The future development directions of dproc include several extensions to the interface, as well as implementing several guiding examples of how application-specific monitoring can improve performance.

We are working on providing 'timed' versions of commands: Currently, all the collected statistics are pushed through a channel. In the future, we intend to add a pull interface which allows a node to specify the maximum age of the information that it is willing to accept, on a per channel basis. If the information that is currently stored in /proc for a particular node is too old, the deamon will pull 'fresh' information. This call is a blocking version of listing the contents of /proc files.

We are also interested in developing large-scale parallel codes that exploit customized monitoring tools. One project under way takes a large-scale parallel password cracking algorithm and develops a customized monitoring infrastructure. Utilizing dproc allows the master in this master-worker program to asynchronously determine the progress and performance of the workers (through remote cache miss statistics) and thereby optimize work allocation. Another project utilizes run queue statistics coupled with caching data to optimize the co-allocation problem of a molecular dynamics simulation with the parallel ray-tracer which performs the visualization.

References

1. "Supporting Efficient Execution in Heterogeneous Distributed Computing Environments with Cactus and Globus". G.Allen, T. Dramlitsch, I. Foster, T. Goodale, N. Karonis, M. Ripeanu, E. Seidel and B. Toonen, Proceedings of SC 2001, November 10-16, 2001.
2. "Distance Visualization: Data Exploration on the Grid". I. Foster, J. Insley, G. vonLaszewski, C. Kesselman, M. Thiebaux, (IEEE Computer Magazine, 32 (12):36-43, 1999).
3. Asmara Afework, Michael Benyon, Fabian E. Bustamante, Angelo DeMarzo, Renato Ferreira, Rovert Miller, Mark Silberman, Joel Saltz, Alan Sussman. "Digital Dynamic Telepathology - the Virtual Microscope", In *Proc. of the 1998 AMIA Annual Fall Symposium*, August, 1998.
4. Fabian Bustamante, Greg Eisenhauer, Karsten Schwan, and Patrick Widener. "Active Streams and the effects of stream specialization". Poster in Proc. of the Tenth IEEE International Symposium on High Performance Distributed Computing-10 (HPDC-10), San Francisco, California, August 6-9, 2001.
5. K. Czajkowski, S. Fitzgerald, I. Foster, C. Kesselman, "Grid Information Services for Distributed Resource Sharing." Procedings of the Tenth IEEE International Symposium on High-Performance Distributed Computing (HPDC-10), IEEE Press, August 2001. Networks, Vol. 2, No. 3, 1999
6. A. DeWitt, T. Gross, B. Lowekamp, N.Miller, P.Steenkiste, J. Subhlok, D. Sutherland, "ReMoS: A Resource Monitoring System for Network-Aware Applications". Carnegie Mellon School of Computer Science, CMU-CS-07-194.

7. Greg Eisenhauer, Weiming Gu, Karsten Schwan and Niru Mallavarupu. "Falcon – Toward Interactive Parallel Programs: The On-line Steering of a Molecular Dynamics Application", In Proceedings of The Third International Symposium on High-Performance Distributed Computing (HPDC-3), San Francisco, August 1994. IEEE Computer Society
8. Greg Eisenhauer, Weiming Gu, Thomas Kindler, Karsten Schwan, Dilma Silva and Jeffrey Vetter. "Opportunities and Tools for Highly Interactive Distributed and Parallel Computing", chapter in Parallel Computer Systems: Performance Instrumentation and Visualization, Rebecca Koskela and Margaret Simmons, editors, ACM Press, 1996.
9. Greg Eisenhauer and Karsten Schwan. "An Object-Based Infrastructure for Program Monitoring and Steering", In Proceedings of the 2nd SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT'98), pp. 10-20, August 1998
10. M.E. Fiuczynski, R.P. Martin, T. Owa, and B.N. Bershad. SPINE: An Operating System for Intelligent Network Adapters. Proceedings of the Eighth ACM SIGOPS European Workshop, pp. 7-12. Sintra, Portugal, September 1998.
11. "A Quality of Service Architecture that Combines Resource Reservation and Application Adaptation". I. Foster, A. Roy, V. Sander, (8th International Workshop on Quality of Service, 2000).
12. Ch. Glasner, R. Hgl, B. Reitinger, D. Kranzlmller, J. Vo. "The Monitoring and Steering Environment" Proc. ICCS 2001, Intl. Conference on Computational Science, San Francisco, CA, USA, pp. 781-790 (May 2001).
13. Hart, Delbert; Kraemer, Eileen; Roman, Gruia-Catalin "Interactive Visual Exploration of Distributed Computations," In Proceedings of 11th International Parallel Processing Symposium, pp.
14. http://ganglia.mrcluster.org/
15. http://smile.cpe.ku.ac.th/software/scms
16. Jeffrey K. Hollingsworth. Finding Bottlenecks in Large-scale Parallel Programs. Ph.D. Dissertation, August 1994. 11-127, Geneva, Switzerland, April 1997
17. Carsten Isert and Karsten Schwan, "ACDS: Adapting Computational Data Streams for High Performance", International Parallel and Distributed Processing Symposium (IPDPS), May 2000.
18. D. Kranzlmller, N. Stankovic, J. Volker"Debugging Parallel Programs with Visual Patterns" Proc. VL'99, 1999 IEEE Symposium on Visual Languages, Tokyo, Japan, pp. 180-181 (Sept. 1999).
19. Robin Kravets, Ken Calvert, and Karsten Schwan, "Payoff-Based Communication Adaptation based on Network Service Availability", IEEE Multimedia Systems '98(ICMCS), Aug. 1998
20. Rajamar Krishnamurthy, Karsten Schwan, and Marcel Rosu, "A Network Co-Processor-Based Approach to Scalable Media Streaming in Servers", International Conference on Parallel Processing (ICPP), August 2000.
21. Beth Plale, Volker Elling, Greg Eisenhauer, Karsten Schwan, Davis King, and Vernard Martin, Realizing Distributed Computational Laboratories, Int'l Journal of Parallel and Distributed Systems and Networks, Vol 2, Num 3, 1999.
22. Randy Ribler, Jeffrey Vetter, Huseyin Simitci, Daniel Reed. "Autopilot: Adaptive Control of Distributed Applications",High Performance Distributed Computing, August, 1999
23. Ariel Tamches, Barton P. Miller.Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels Operating Systems Design and Implementation, 1999

24. TotalView Monitoring software, Etnus LLC. http://www.etnus.com.
25. Jeffrey Vetter and Karsten Schwan, "Techniques for High Performance Computational Steering", IEEE Concurrency, Oct-Dec 1999.
26. Richard West and Karsten Schwan, "Quality Events: A Flexible Mechanism for Quality of Service Management," in Proceedings of the 7th IEEE Real-Time Technology and Applications Symposium (RTAS), 2001.
27. Patrick Widener, Karsten Schwan, and Greg Eisenhauer. "Open Metadata Formats: Efficient XML-Based Communication for Heterogeneous Distributed Systems". Proceedings of the Poster Session of the 21st IEEE International Conference on Distributed Computing Systems, Phoenix (Mesa), Arizona, April 16-19, 2001.
28. Rich Wolski, Neil Spring, and Jim Hayes."The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing", Journal of Future Generation Computing Systems, 1998