TAUoverSupermon: Low-Overhead Online Parallel Performance Monitoring

Aroon Nataraj[1], Matthew Sottile[2], Alan Morris[1], Allen D. Malony[1], and Sameer Shende[1]

[1]Department of Computer and Information Science
University of Oregon, Eugene, OR, USA,
{anataraj,amorris,malony,sameer}@cs.uoregon.edu

[2]Los Alamos National Laboratory, Los Alamos, NM, USA,
matt@lanl.gov

Abstract. Online application performance monitoring allows tracking performance characteristics during execution as opposed to doing so post-mortem. This opens up several possibilities otherwise unavailable such as real-time visualization and application performance steering that can be useful in the context of long-running applications. As HPC systems grow in size and complexity, the key challenge is to keep the online performance monitor scalable and low overhead while still providing a useful performance reporting capability. Two fundamental components that constitute such a performance monitor are the measurement and transport systems. We adapt and combine two existing, mature systems - TAU and Supermon - to address this problem. TAU performs the measurement while Supermon is used to collect the distributed measurement state. Our experiments show that this novel approach leads to very low-overhead application monitoring as well as other benefits unavailable from using a transport such as NFS.
Keywords: Online performance measurement, cluster monitoring.

1 Introduction

Online or real-time application performance monitoring tracks performance characteristics during execution and makes that information available to consumers at runtime. In contrast to post-mortem performance data analysis, developing monitoring capabilities opens up several possibilities otherwise unavailable, such as real-time visualization and application performance steering. One advantage of performance monitoring can be found in the tracking, adaptation, and control of long-running applications. The simple ability to detect problems early in a running application and terminating the execution has practical benefits to savings in computing resource usage. However, as HPC systems grow in size and complexity, building efficient, scalable parallel performance monitoring systems that minimize adverse impact on applications is a challenge.

Two fundamental components constitute an online application performance monitor: 1) the performance measurement system and 2) the transport system.

The infrastructure for performance measurement defines what performance metrics can be captured for events in individual execution contexts (e.g., processes and threads). Effectively, the measurement system is the performance data producer. The transport system enables querying the parallel/distributed performance state from the different contexts and delivers the data to monitoring consumers. The transport system acts as a bridge between where the data is generated to where it is consumed, but it can also be used to control the measurement system to control type and rate of performance data production.

While any performance measurement raises issues of overhead and perturbation, online monitoring introduces additional concerns. In static application performance measurement (with post-mortem data access and analysis), the measurement sub-systems of individual contexts are isolated as they perform local measurements and are impacted little by the scale of the parallel application. In contrast, the need to retrieve and integrate the global performance state of individual contexts means measurement sub-systems are no longer isolated and must support periodic interactions from the monitor, potentially affecting performance behavior and scalability. The challenge is to create an online measurement system that is scalable and very low overhead and can still provide useful performance analysis capabilities.

Our solution, *TAUoverSupermon* (*ToS*), adapts two existing mature systems – TAU [13] (as the measurement system) and Supermon [14] (as the transport system) – to address the problem of scalable, low-overhead online performance monitoring of parallel applications. We describe the ToS design and architecture in Section 3. The efficiency of our approach is demonstrated in Section 4 through experiments to evaluate *ToS* performance and scalability. An example to demonstrate application/system performance correlation is provided in Section 5. Section 6 examines related work. The paper concludes in Section 7 with final remarks and a brief discussion of future directions. We begin with a discussion of the rationale behind our approach in the next section, Section 2.

2 Rationale and Approach

There are several approaches to building a parallel application monitoring framework, combining existing performance measurement infrastructure with a system to transport performance data to monitoring consumers. The simplest is to use filesystem capabilities for data movement. Here, the performance measurement layer implements operations to write performance data to files. The filesystem aggregates the performance information which can be seen by consumers when they perform file read operations. Concurrent file I/O between the parallel application and performance data consumers must be possible for this alternative to be of any use. While NFS support provides for robust implementation of this approach, file I/O overheads can be high (both on the producer and consumer sides), the monitor control must also be implemented through file transactions, and NFS has known scalability issues. Furthermore, in the rare case that a glob-

ally visible filesystem is unavailable, a file-based transport system is simply not an option.

If a file system is not used, some alternative transport facility is necessary. A measurement system could be extended to include this support, but it would quickly lead to incompatibilities between monitors and significant programming investment required to build a scalable, low-overhead transport layer. Instead, where transport functionality has been developed in other system-level monitoring tools, an application-level performance monitor can leverage the functionality for performance data transport. Our approach couples the Supermon cluster monitor with the TAU application measurement system to provide the measurement and transport capabilities required for online parallel application performance monitoring. TAU performs the role of the performance producer (source) and is adapted to use Supermon as the transport from which consumers (sinks) query the distributed performance state.

The rationale behind TAUoverSupermon is based on the following criteria:

Reduced overhead. Using a traditional filesystem as monitor transport incurs high overhead. Currently, TAU allows runtime performance output through the filesystem, but suffers high overheads as discussed in Section 4.

Autonomous operation. Keeping the transport outside TAU makes available an online transport facility to any other system components that want to use it. Supermon can be used for several purposes, such as its default system monitoring role, independently of TAU.

Separation of concerns. Concerns such as portability, scalability, and robustness can be considered by the measurement and transport systems separately. Both TAU and Supermon are mature, standalone systems whose implementations are optimized for their respective purposes.

Performance correlation. This approach allows close correlation between system level information (such as from the OS and hardware sensors) with application performance (see Section 5). This facilitates determining the root cause of performance problems that may originate from outside of the application.

Light-weight control. Feedback to the measurement system is important for controlling monitoring system overhead. The control path should be light weight to avoid having its use be a significant contributing factor.

3 The *TAUoverSupermon* Architecture

The high-level architecture of ToS is shown in Figure 1. It is composed of the following interacting components:

TAU. The TAU measurement system (in blue) generates performance data for each thread of execution due to application instrumentation. TAU implements an API to output performance data during or at the end of execution.

Supermon. The Supermon transport, including the root and intermediate Supermon daemons are located on intermediate (or service) nodes. The mon

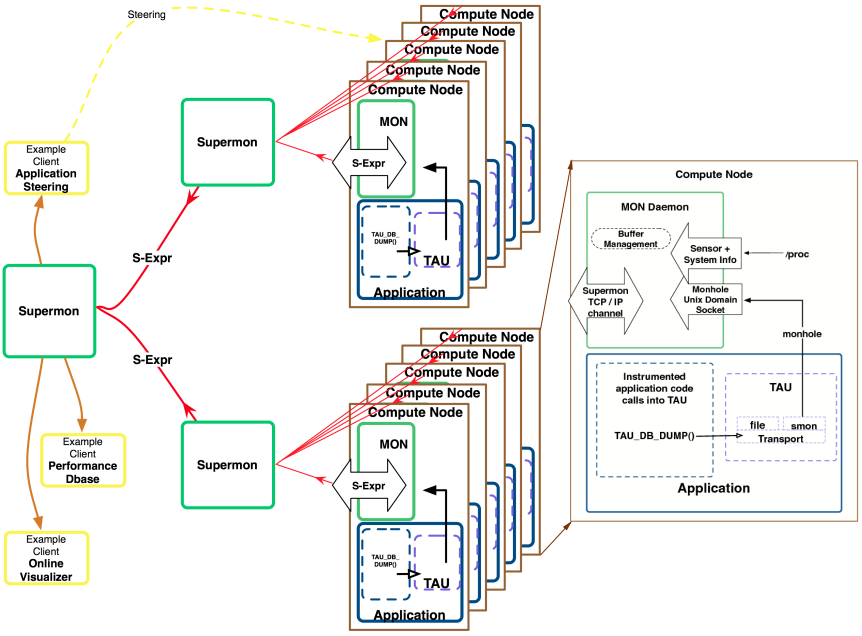

Fig. 1. *TAUoverSupermon* Architecture

daemons (shown in green), are located on each compute node where the application is running.

Clients. Monitor clients (sinks) of the online performance data (shown in yellow) can be located anywhere on the network connected to the cluster.

In the following sections we discuss each of the components in detail, their implementation, and the changes we made to adapt them for *ToS*.

3.1 TAU Performance System

TAU [13] is an integrated toolkit for parallel performance instrumentation, measurement, analysis, and visualization of large-scale parallel applications. TAU provides robust support for multi-threaded, message passing, or mixed-mode programs. Multiple instrumentation options are available, including automatic source instrumentation. The TAU measurement system implements both parallel profiling and tracing. We are mainly concerned with profile-based performance data in this paper. TAU supports an API to read the performance profile data stored in its internal data structures at runtime, and to output the performance data to a monitoring system using the TAU DB DUMP() call. It is this output interface that will be extended to support Supermon.

3.2 Supermon Cluster Monitor

Supermon [14] is a scalable monitoring system for high performance computing systems. Its current implementation includes a set of socket-based servers that gather, organize and transfer monitoring data in a format based on LISP symbolic expressions (called s-exprs). Its architecture is hierarchical, whereby each node is treated as a leaf node in a tree, while a series of data concentrators gather data from the nodes and transport it to a root. The root then provides data to clients. Supermon's primary purpose has been in monitoring system-level performance such as those reported by hardware-sensors and OS performance data. The Supermon architecture builds on prior experiences with prior implementations based on SunRPC, followed by a non-hierarchical wire protocol over sockets, to achieve a low-overhead, highly extensible monitoring system in the current design. A Supermon system consists of the following components:

Kernel module. Within the compute node OS is a kernel module that exports system-level parameters locally as a file in the */proc* pseudo-filesystem formatted as s-exprs to avoid overhead from parsing the inconsistent formats of the various data sources.

***mon* daemon.** The *mon* daemon, on each compute node, reads the file under */proc* and makes available the system-level metrics as s-exprs over the network via TCP/IP.

***monhole*.** *mon* also listens on a Unix Domain Socket (UDS) locally accepting data from any other sources outside the kernel. This interface to *mon* is called the *monhole*.

Supermon daemon. On the service node(s), there is the Supermon daemon. This talks to each of the *mon* daemons and queries and collects the data from them. This data includes the */proc* system-level parameters as well as data submitted through the *monhole* interface. Supermon then makes this data available as another s-expression to any interested clients. Using the same s-expr format as the *mon* daemons, Supermon daemons may act as clients to other Supermon daemons to create a scalable, hierarchical structure for transporting data from sources to sinks.

3.3 Coupling TAU and Supermon

Figure 1 depicts the interaction between the application instrumented with TAU and the *mon* daemon on the compute node through the *monhole* interface. Below we describe the changes made to Supermon and TAU to build the *ToS* system.

Adapting Supermon

The *mon* daemon provides the UDS-based *monhole* interface for external sources to inject data. We tested and updated *monhole* for TAU's use and made its buffering characteristics more flexible. The buffering policy name refers to how the existing data in the *mon* daemon buffer is managed due to a new write or a read. Some possible policies are:

***REPLACE-READ*:** Existing data is replaced (i.e., overwritten) on a write, and the buffer remains unaffected on a read.
***FILL-DRAIN*:** Writes append to buffer and reads empty the buffer.
***REPLACE-DRAIN*:** Writes replace buffer data and reads empty the buffer.
***FILL(K)-READ*:** Writes append data, but the buffer is unaffected by reads. Given a ring buffer of finite size K, repeated writes ($> K$) will overwrite data.

The buffer policy is important as it determines how the buffer is affected by multiple concurrent clients and what data is retrieved. It also determines what guarantees regarding data loss can be made (e.g., when sampling rate does not match data generation rate), and what memory overhead on the data source is required to support maintaining data for clients to read.

Initially, the *monhole* only supported the simple and efficient *REPLACE-READ*. This policy has several advantages: i) slow sinks will not cause infinitely large buffers to be maintained at the *mon* daemon, and ii) multiple sinks can query the data simultaneously without race conditions. However, the policy suffers from potential data-loss when sink read rate (even transiently) is less than source generation rate, or bandwidth waste when sinks query too frequently and receive the same data. A small configurable buffer under *FILL(K)-READ* can alleviate the former, whereas a *REPLACE-DRAIN* strategy can remedy the latter when a single client is used. For these reasons, we implemented a runtime-configurable buffer strategy. The repair mechanism for hierarchical topologies was also fixed in Supermon.

Adapting TAU

Prior to our work with Supermon, TAU assumed the presence of a shared network filesystem for performance monitoring. Buffered file I/O routines were used in the TAU monitoring API. We first made the notion of *transport* a first-class entity by creating a generic transport class. To keep changes isolated to a small portion of the TAU code base, the generic transport class needed to expose interfaces exactly like the file I/O calls in the standard I/O library, *stdlib*. As shown in Figure 1, two implementations of this transport class were created: one for the default *stdlib* file I/O and the other for use with the *monhole* interface. The type and nature of the transport being used is kept hidden from the TAU API. The type of transport can be fixed statically at compile-time or can be communicated to the application via an environment variable at application startup. While read/write operations on the *monhole* are performed directly, other operations such as directory creation are not directly available and need to be forwarded to sinks (on control nodes). This framework allows easy extension by adding new custom transports to TAU in the future.

4 Investigating Performance and Scalability

To evaluate TAUoverSupermon we use the NAS Parallel LU application (Class C) benchmark [1] instrumented with TAU under different configurations. The

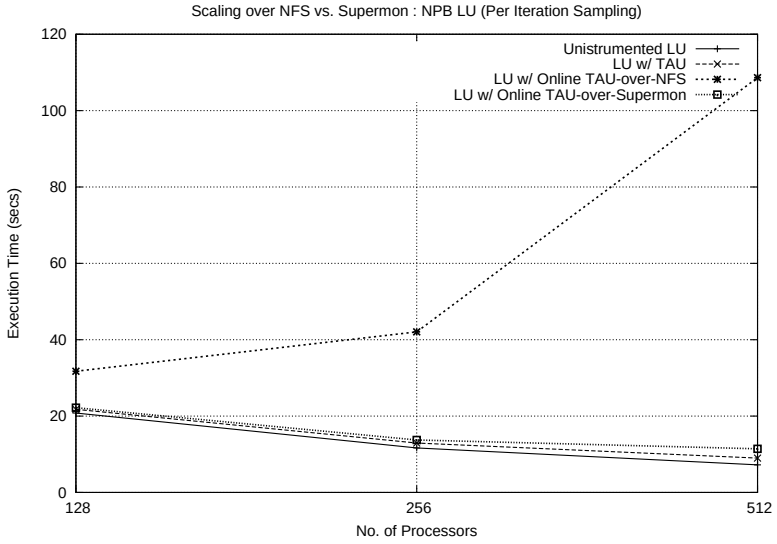

Fig. 2. Execution Time and Overhead

choice of the benchmark was guided by the need for a representative parallel workload; one that triggers a sufficient number of events, so as to study the overhead generated as a function of number of profile measurements that take place. LU has a mix of routine and MPI events and an understood parallel algorithm that lets us relate overhead to scaling behavior. We compare the performance of NPB LU under the following configurations:

LU-none Basic LU without instrumentation.
LU-PM LU instrumented with TAU for post-mortem measurement data.
LU-NFS LU instrumented with TAU for online measurement using NFS.
LU-ToS LU instrumented with TAU for online measurement using Supermon.

Online measurement data retrieval of LU is performed at a frequency of *once per iteration*. We repeat each of the runs over 128, 256 and 512 nodes to examine scalability. The Atlas cluster from Lawrence Livermore National Lab, with quad dual-core Opteron Linux nodes running Infiniband, serves as our test environment. The metrics we use are the total runtime reported by LU and the overhead as % dilation, computed as the total runtime under some configuration divided by the total runtime of *LU-none* configuration.

In Figure 2 we plot the runtime of the LU benchmark under the different configurations as the processor count increases. The following observations are clear:

– TAU measurements (LU-PM) contributed 4.7% (N=128) to 24.6% (N=512) overhead. Re-runing the LU-PM (N=512) with TAU configured to use the light weight cycle counter (*rdtsc*) for timing brought the overhead down to just 2.5%.
– Overhead of online performance measurement and data-retrieval using NFS is at least 52.71% and grows super-linearly as the number of CPUs increase to a staggering 1402.6%.
– Overhead of online performance measurement and data-retrieval using Supermon is close to the TAU overhead of post-mortem data retrieval (as low as 6.83%).
– As LU scales, the savings obtained from using Supermon transport as opposed to NFS grow super-linearly.

It is remarkable that, for the test measurement and sampling rate, online measurement with *ToS* can be provided nearly for free over the cost of the *post-mortem* run. We also ran experiments for the 128 node case (Class B) on the MCR cluster at Lawrence Livermore National Laboratory. There the following dilations were observed: LU-PM 8.5%, LU-NFS 72.6% and LU-ToS 9.1%.

Type	**rename**	**select**	**open**	**writev**	**read**	**close**	**write**
Tau-NFS	**11.75**	9.46	**8.55**	4.02	3.22	2.50	0.63
Tau-PM	**0**	5.94	**0.03**	3.95	3.22	0	0.60

Table 1. Comparing System Calls: Online TAU-NFS vs. Postmortem TAU (secs)

Why is there such a dramatic difference in performance between using the NFS transport and Supermon? To further investigate what aspects of the system contribute to the significant savings accrued, we use KTAU [9] to measure kernel-level events. Smaller LU-PM and LU-NFS experiments on 4 nodes (of Pentium III dual-CPU over Ethernet) are run, this time under a KTAU kernel. Table 1 compares the runtime of the largest system calls under both configurations, as measured by KTAU. Surprisingly the largest differences are seen in `sys_rename` and `sys_open` and not in the read/write calls. Why?

When files are used to transport performance data from TAU to a monitoring client, there is a problem of read consistency. If the client polls for new data, how does it know when and how much data is new? TAU uses a two-stage process: 1) write to a temporary file, then 2) rename the file to the filename being polled by the client. This approach employs the `rename` and `open` meta-data operations on every performance data dump. These meta-data operations are synchronous and blocking (between the client and the server), unlike the buffered read/write operations in NFS. The impact of these simultaneous meta-data operations grows significantly as node-count increases. In the case of the Supermon transport, these operations are not performed locally. Instead they are also made asynchronous (non-blocking) and performed by the sink (on the control/head node).

Another aspect to note is the *'per iteration sampling frequency'* used (instead of, say, a fixed *1Hz sampling*). Because of the strong scaling nature of LU, as the number of nodes increase, the iterations become shorter and the overhead per unit time from data retrieval increases. When the dump operation is relatively costly, as in NFS, it results in the superlinear scaling behavior. In addition, the variability in the time taken by each NFS dump operation across the ranks leads to magnification of the overhead.

5 Online Application/System Performance Correlation

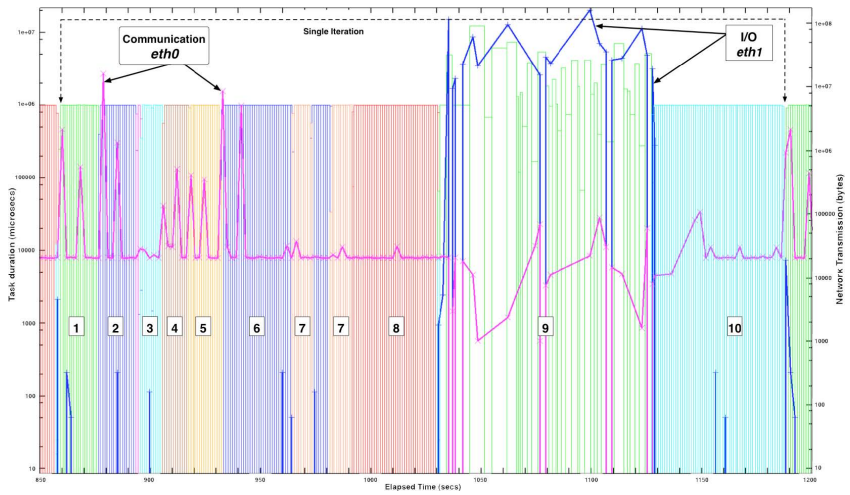

Fig. 3. Correlating Uintah Phases with System-level metrics

To give a sense of the power of online performance monitoring, we report results from a performance investigation of the Uintah Computational Framework [2] where the application performance is correlated with runtime system actions. Figure 3 shows the execution timeline of a single iteration of an Uintah application (*bigbar* using the Material Point Method) where performance data from 1 Hz monitoring is plotted. The performance data is coming from two sources: the application and the system-level monitoring. Both sources used Supermon for the transport and the data streams were available as separate s-expressions on the Supermon channel. The execution took place on a 32-processor Pentium Xeon Linux cluster in our lab. The cluster is served by two separate physical networks, one providing connectivity among back-end compute nodes (through interface eth0) and another providing NFS connectivity (through eth1).

What stands out in the figure are the phases of the application's computation and the correlated network demands on the two separate interfaces. The phases

are numbered (and distinctly colored) so as to differentiate them. On the xaxis is the time elapsed since the start of the application (the iteration shown falls between 850 and 1200 seconds). The left y-axis plots the difference in task duration between consecutive samples. On the right y-axis are plotted differences in bytes transmitted between samples and this is overlayed on the application phases as two solid lines - magenta for interface eth0 and blue for interface eth1.

For each monitoring interval, the profile sample is drawn to show the performance data for the dominant events. These phases would not be apparent if the profile data was not sampled periodically by the monitor. In this way, application and system performance can be correlated to better understand runtime effects. For instance, the impact of MPI and checkpoint operations on communication and I/O are clearly apparent. Tasks 1 through 8 mostly perform communication (seen from the eth0 curve), whereas task 9 (which is checkpointing) performs I/O to NFS (over eth1). Then Task 11 (MPI_Allreduce) ends the iteration. This correlation would be infeasible by direct measurement from within the application alone as it is unaware of system-level factors (e.g. the network topology and interfaces exercised).

6 Related Work

TAUoverSupermon owes its heritage to a long line of online performance monitoring projects. On-line automated computational steering frameworks like Falcon [16], Autopilot [10], Active Harmony [15], and MOSS [3] use a distributed system of sensors to collect data about an application's behavior and actuators to make modifications to application variables. These systems have built-in transport support and require the application to be modified to expose steerable parameters. In contrast, *ToS* couples two independent, standalone systems, and builds on a lower-level interface between TAU and Supermon which allows for more flexibility in its specific use. While we have not applied *ToS* to steering, we have demonstrated measurement control with Supermon using reverse channels supported in the *monhole*. It is conceivable that higher-level methods provided by these tools could also be layered on *ToS*.

It is important to distinguish between monitoring systems intended for *introspective* versus *extrospective* use. Scalability and low overhead for global performance access is important for introspective monitoring. Paradyn's Distributed Performance Consultant [8] supports introspective online performance diagnosis and uses a high-performance data transport and reduction system, MRNet [11], to address scalability issues [12]. Our TAUg [5] project demonstrated scalable, online global performance data access for application-level consumption by building access and transport capabilities in a MPI library linked with the application. On the other hand, monitoring systems to be used by external clients require support for efficient network communications, in addition to source monitoring scalability. The On-line Monitoring Interface Specification (OMIS) [6] and the OMIS compliant monitoring (OCM) [17] system target the problem of providing a universal interface between online, external tools and a monitoring system.

OMIS supports an *event-action* paradigm to map events to requests and responses to actions, and OCM implements a distributed client-server system for these monitoring services. However, the scalability of the monitoring sources and their efficient channeling to off-system clients are not the primary problems considered by the OMIS/OCM project.

Fürlinger and Gerndt's work on Periscope [4] addresses both the scalability and external access problems by using hierarchical monitoring agents executing in concert with the application and client. The agents are configured to implement data reduction and evaluate performance properties, routing the results to interactive clients for use in performance diagnosis and steering. MRNet can also be used for extrospective monitoring. It is organized as a hierarchy of processes, created separately from the application processes, allowing it to connect to remote monitor sinks. Like MRNet-based tools, TAU can use Supermon in a flexible and scalable manner for both introspective and extrospective monitoring. The *ToS* work reported here demonstrates this performance monitoring functionality. It also shows how the *ToS* approach imposes few reengineering requirements on the monitoring sources and clients, allowing for a clean, light-weight implementation. It is interesting to note, that we could build a *TAUoverMRNet* monitoring system, and have plans in this regard.

7 Conclusions and Future Work

The desire to perform very-low overhead online application performance measurement led us to investigate alternatives to the traditional *'store performance data to shared-filesystem'* approach. We created a large-scale online application performance monitor by using Supermon as the underlying transport for the TAU measurement system. Experiments demonstrate that the *TAUoverSupermon* solution provides significantly lower overhead and greater scalability. Another demonstrated advantage to using an existing cluster-monitor as the transport is that it allows close correlation of application performance with system-level performance information. This facilitates separating performance effects that originate from within an application and those that are due to external effects outside the control of the application itself.

The scalability of a parallel performance monitoring system depends on several factors related to how it is designed and engineered as well as to how the system is used. Here we have demonstrated reduction in overheads for source data generation and transport. We are also experimenting with strategies to improve scalability further by reducing the number of nodes touched per query (e.g., using sampling [7]) and/or by reducing the data generated per node per query through aggregation. By having greater control over the transport and by being able to add extra intelligence into it, the *ToS* system can allow easy implementation of the above strategies. Other directions along which we would like to take this work include experimentation on very large scale platforms such as BG/L (already ported and functional), and adding new custom transports to TAU such as MRNET.

References

1. D. H. Bailey et. al. The nas parallel benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.
2. J. D. de St. Germain, S. G. Parker, J. McCorquodale, and C. R. Johnson. Uintah: A massively parallel problem solving environment. In *HPDC'00: International Symposium on High Performance Distributed Computing*, pages 33–42, 2000.
3. G. Eisenhauer and K. Schwan. An object-based infrastructure for program monitoring and steering. In *2nd SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT'98)*, pages 10–20, 1998.
4. M. Gerndt, K. Fürlinger, and E. Kereku. Periscope: Advanced techniques for performance analysis. In *Parallel Computing: Current & Future Issues of High-End Computing, In the International Conference ParCo 2005, 13-16 September 2005, Department of Computer Architecture, University of Malaga, Spain*, pages 15–26, 2005.
5. K. A. Huck, A. D. Malony, S. Shende, and A. Morris. TAUg: Runtime Global Performance Data Access Using MPI. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 4192/2006 of *Lecture Notes in Computer Science*, pages 313–321, Bonn, Germany, 2006. Springer Berlin / Heidelberg.
6. T. Ludwig, R. Wismüller, V. Sunderam, and A. Bode. Omis – on-line monitoring interface specification (version 2.0). *LRR-TUM Research Report Series*, 9, 1998.
7. C. Mendes and D. Reed. Monitoring large systems via statistical sampling. *International Journal of High Performance Computing Applications*, 18(2):267–277, May 2004.
8. B. Miller, M. Callaghan, J. Cargille, J. Hollingsworth, R. Irvin, K. Karavanic, K. Kunchithapadam, and T. Newhall. The paradyn parallel performance measurement tool. *Computer*, 28(11):37–46, 1995.
9. A. Nataraj, A. Malony, S. Shende, and A. Morris. Kernel-Level Measurement for Integrated Parallel Performance Views: the KTAU Project. In *CLUSTER'06: International Conference on Cluster Computing*. IEEE Computer Society, 2006.
10. R. Ribler, H. Simitci, and D. Reed. The Autopilot performance-directed adaptive control system. *Future Generation Computer Systems*, 18(1):175–187, 2001.
11. P. Roth, D. Arnold, and B. Miller. Mrnet: A software-based multicast/reduction network for scalable tools. In *SC'03: ACM/IEEE conference on Supercomputing*, 2003.
12. P. Roth and B. Miller. On-line automated performance diagnosis on thousands of processes. In *11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 69–80, 2006.
13. S. Shende and A. D. Malony. The TAU parallel performance system. *The International Journal of High Performance Computing Applications*, 20(2):287–331, Summer 2006.
14. M. Sottile and R. Minnich. Supermon: A high-speed cluster monitoring system. In *CLUSTER'02: International Conference on Cluster Computing*, 2002.
15. C. Tapus, I.-H. Chung, and J. Hollingworth. Active harmony: Towards automated performance tuning. In *SC'02: ACM/IEEE conference on Supercomputing*, 2002.
16. W. Gu et. al. Falcon: On-line monitoring and steering of large-scale parallel programs. In *5th Symposium of the Frontiers of Massively Parallel Computing, McLean, VA,*, pages 422–429, 1995.
17. R. Wismuller, J. Trinitis, and T. Ludwig. Ocm – a monitoring system for interoperable tools. In *2nd SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT'98)*, pages 1–9, 1998.