A Framework for Scalable, Parallel Performance Monitoring using TAU and MRNet

Aroon Nataraj[1] Allen D. Malony[1] Alan Morris[1] Dorian Arnold[2]
Barton Miller[2]

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

[2]Computer Sciences Department
University of Wisconsin, Madison, WI, USA
{darnold,bart}@cs.wisc.edu

Abstract

Performance monitoring of HPC applications offers opportunities for adaptive optimization based on dynamic performance behavior, unavailable in purely post-mortem performance views. However, a parallel performance monitoring system must have low overhead and high efficiency to make these opportunities tangible. We describe a scalable parallel performance monitor called TAUoverMRNet (ToM), created from the integration of the TAU performance system and the Multicast Reduction Network (MRNet). The integration is achieved through a plug-in architecture in TAU that allows selection of different transport substrates to offload online performance data. A method to establish the transport overlay structure of the monitor from within TAU, one that requires no added support from the job manager or application, is presented. We demonstrate the distribution of performance analysis from the sink to the overlay nodes and the reduction in large-scale profile data that could otherwise overwhelm any single sink. Results show low perturbation and significant savings accrued from reduction at large processor-counts.

Keywords: *Online, measurement, reduction*

1 Introduction

With the advent of multi-core, heterogeneous, and extreme scale parallel computing, there has been a recent shift in perspective [1] of parallel performance analysis as a purely static, offline process to one requiring online support for dynamic monitoring and adaptive performance optimization. Given the prerequisites of low overhead and low perturbation for performance measurement methods, the addition of runtime performance query and analysis capabilities would seem antithetical to the performance tool orthodoxy. What is surprising, however, is the willingness in the *neo-performance* perspective to consider the allocation of additional system resources to make dynamic performance-driven optimization viable. Indeed, as parallel systems grow in complexity and scale, this may be the only way to reach optimal performance.

A *parallel performance monitor* couples a system for performance measurement with runtime infrastructure for accessing performance data during program execution. Parallel performance measurement systems, such as the TAU Performance System™[2], can scale efficiently by keeping performance data local to where threads of execution are measured. Providing low-overhead access to the execution-time performance data for dynamic analysis

is a different challenge because it requires global program interaction. If additional system resources can be utilized, a robust parallel performance monitor can be built.

How performance monitoring is used in practice (e.g., frequency of interaction, amount of performance data, # processors) will define architectural guidelines for a monitor's design. However, to optimize the tradeoff of monitoring overhead versus additional resource assignment, a comprehensive characterization of monitor operation is required. It is important to provide a flexible framework for scalable monitoring and a methodology for evaluation that would allow engineering optimizations to be determined given choices of acceptable levels of overhead and resource allocation.

The *TAU over MRNet (ToM)* performance monitor integrates TAU with the MRNet scalable infrastructure for runtime program interaction. This paper reports our experiences building a scalable parallel monitor (based on the *ToM* prototype) and evaluating its function and performance. Section 2 presents the system design and operational model. Here we define an abstract monitoring interface to support infrastructure interoperability and leverage MRNet's programming capabilities for analysis-filter development. Section 3 describes how the transport network is instantiated at the start of program execution. Once in place, *ToM* can be used in a variety of ways for performance data analysis. Section 4 discusses the different methods we have implemented. In Section 5 we assess monitor performance using benchmark codes and the FLASH application. Our goal is to evaluate different parameters of *ToM*'s configuration and use. Given space constraints, we refer the reader to the related work described in our first-generation work on performance monitoring [3] and our MRNet paper [4].

2 Scalable Monitor Design

The problem of scalable, online parallel monitoring naturally decomposes into measurement and transport concerns. With TAU performing the measurement, the choice of a transport needs to consider several factors including, i) the availability of specialized physical networks, ii) the nature and size of performance data (e.g. profile vs. trace) and feasibility of distributed analyses and data reduction, iii) availability of monitoring/transport resources and iv) the perturbation constraints. An extensible plugin-based architecture that allows composition of the measurement system with multiple, different transports allows flexibility in the choice of a transport based on these factors. Our current work, an extension of *TAUoverSupermon* [3], generalizes that approach and goes further in exploring the Tree-Based Overlay Network (TBON) model with an emphasis on programmability of the transport (distributed analysis/reduction) and a transparent solution to allocation of transport/application resources. We demonstrate scalable monitor design using the *TAUoverMRNet* (*ToM*) prototype. The main components and the data/control paths of the system (shown in Figure 1) are described next.

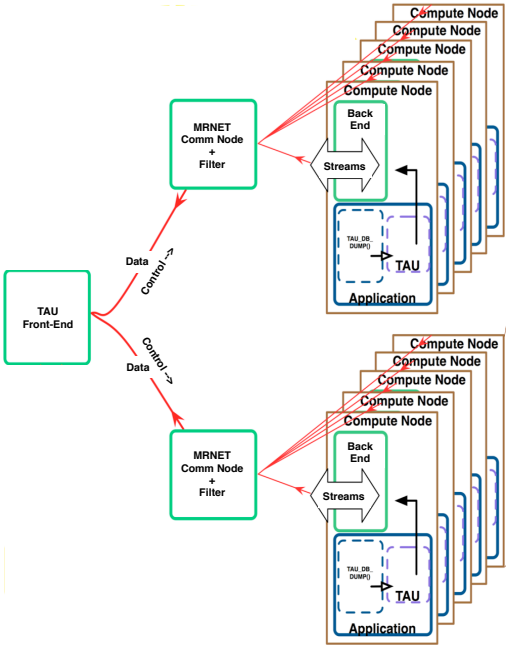

Figure 1. The *TAUoverMRNet* System

2.1 Back-End

Figure 2 shows the currently available bindings (NFS, Supermon and MRNet). The profile output routine (TAU_DB_DUMP) in TAU uses a generic interface which is implemented by each of the specific transport adaptors. In the case of a NFS, the implementation directly falls

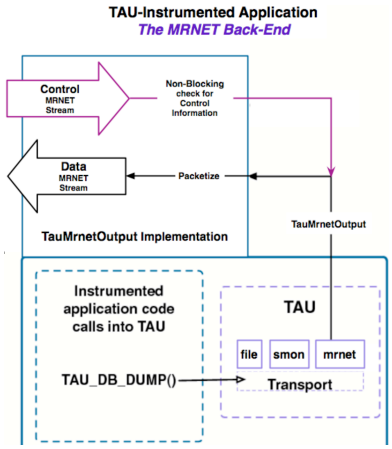

Figure 2. *ToM* **Backend**

through to the standard library implementation of FILE I/O. The choice of the transport can be made at runtime using environment variables. The MRNet adapter in TAU uses two streams, one each for data and control. The data stream is used to send packetized profiles from the application backends to the sink (monitor). The offloading of profile data is based on a *push-pull* model, wherein the instrumented applications push data into the transport, which is in turn drained out by the monitor. The application offloads profile information at application-specific points (such as every iteration) or at periodic timer intervals. The control stream is meant to provide a reverse channel from monitor to application ranks. It is polled to check for control messages on every invocation of the TAU_DB_DUMP() routine. Control traffic includes messages for startup/finalization of transport and to set measurement/instrumentation options.

2.2 Front-End

The *ToM front-end* (*FE*) invokes the MRNet API to instantiate the network and the streams (data, control). It sends an *INIT* on both streams to the application backends (*BE*) allowing the *BE*s to discover the streams. In the simplest case, the data from the application is transported as-is, without transformations. The *FE* continues to receive data until reception of a *FIN* from every *BE*. It then broadcasts a *FINACK* and proceeds to destroy the MRNet network. The simplest *FE* just unpacks and writes the profiles to disk. More sophisticated *FE*saccept and interpret statistical data including histograms and functionally-classified profiles. These *FE*s are associated with special intermediate filters.

2.3 Filters

MRNet provides the capability to perform transformations on the data as it travels through intermediate nodes in the transport topology. *ToM* uses this capability to i) distribute statistical analyses traditionally performed at the sink and ii) to reduce the amount of performance data that reaches the monitor. *UpStream* filters (*USF*) can intercept data going from Back-Ends (*BE*) to the Front-End (*FE*) and DownStream filters (*DSF*) intercept data flowing in the opposite direction. We discuss the use of filtering in *ToM* including distributed histogramming and functional classification in Section 4.

3 Monitor Transport Instantiation

A scalable monitor design that utilizes additional tool-specific resources raises the issue of effectively co-allocating nodes required for efficient transport and analysis alongside the primary nodes of the target application. Given two classes (transport and application), the nodes must be made aware of their roles and the identities of their neighbors in the network topology. To function correctly the system requires that i) an additional set of nodes to be allocated for the purposes of transport and analysis, ii) the topology of the transport be constructed correctly from the allocated set of nodes, iii) the application backends discover and connect to their respective parents in the transport topology and iv) importantly, these requirements are met transparently to both the application and the job scheduling system.

We provide a transparent method for transport instantiation in MPI programs that takes advantage of the fact that TAU intercepts MPI calls (using the PMPI interface) for measurement purposes. In the context of the *ToM* prototype, the steps taken by Rank-0, other tree-ranks and application ranks are listed in Figure 3. When an application calls *MPI_Init()*, TAU intercepts the call on all of the nodes and first calls *PMPI_Init()*. Based on the parameters of the transport topology and the number of nodes in

the application, the required number of transport nodes is calculated. New communicators are created by splitting *COMM_WORLD* into transport (*tomCOMM*) and application (*userCOMM*) ranks.

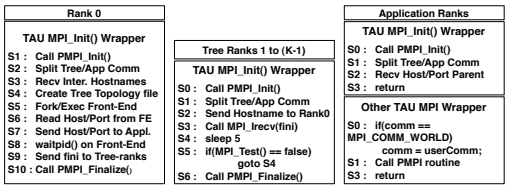

Figure 3. Transport Instantiation

The tree ranks (still within *MPI_Init()*) register their hostnames with Rank-0 using MPI communication, which constructs a topology file and spawns a *ToM FE*. The *FE* in turn uses the MRNet API to instantiate a tree-network and provides to Rank-0 the list of hosts and ports that the application *BE*s need to connect to. Rank-0 sends this information to each *BE* rank so it can connect to the transport. Rank-0 then waits for the *FE*'s termination. The *BE*s return from *MPI_Init()* and execute the rest of the application. Every MPI call from an application rank (*BE*) on the *COMM_WORLD* communicator is intercepted by TAU and the corresponding PMPI call is issued by TAU using the *userCOMM* in place of *COMM_WORLD*. This ensures that no changes are required to the application.

At this stage all the other intermediate tree-ranks could proceed directly to Finalize(). But on many user-level networking solutions (e.g. using Infiniband [5]) blocking MPI calls (like *finalize, barrier, recv, and wait*) poll continuously to avoid context-switch latency. Hence calling *MPI_Finalize()* would consume 100% cpu, starving the intermediate transport processes. To prevent this the intermediate ranks repeatedly perform a non-blocking check for a *FINI* from Rank-0 using *MPI_Irecv*, *MPI_Test* and *sleep* calls. The tree-ranks never return from *MPI_Init()* and instead call *PMPI_Finalize()* inside the TAU *MPI_Init()* wrapper once the *FINI* is received.

4 Distributed Analysis and Reduction

Ideally, one would want to retrieve and store as much performance detail as the measurement system can provide. But the perturbation caused to the measured application and the transport and storage costs associated with the performance data, require that we trade-off measurement data granularity (in terms of events, time intervals and application ranks) against the costs. One method to vary the level of performance detail is through performance data reduction as the data flows through the transport. This is feasible by distributing performance analyses traditionally performed at the front-end, out to the intermediate transport nodes. *ToM* implements three such filtering schemes, each building upon and extending the previous one. Figure 4 describes the data paths used by the distributed analyses and reductions we examine next.

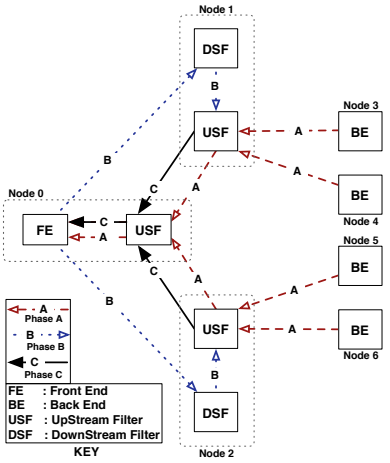

Figure 4. Distributed Analysis

4.1 Statistical Filter

The *StatsFilter*, the simplest *ToM* filter, is an Upstream Filter (*USF*) that calculates global summary statistics across the ranks including mean, standard deviation, maximum and minimum for every event in the profile. Performance data is assumed to arrive in *rounds* (i.e. a round is one profile-offload from every rank). The summary statistics are calculated by an intermediate node over the data from all its children. The resulting measures are passed up to its parent which in turn calculates the measures over the data from all its children and so on until a single set of statistical measures for each event arrives at the monitor. This corresponds to *Phase A* of the data path in Figure 4. The *StatsFilter* consists of a front-end

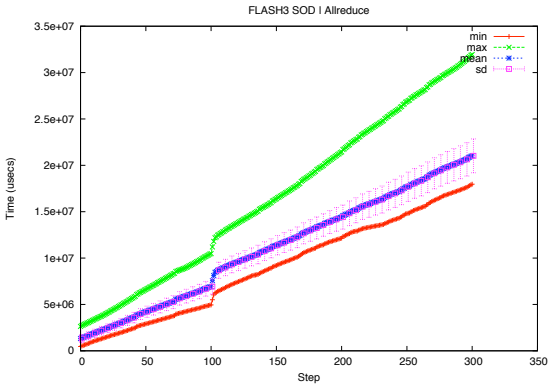

Figure 5. Allreduce Statistics

Stats_FE, derived from *ToM_FE* and a filter shared object *StatsFilter.so* loaded on the intermediate nodes. An example of the output from such a reduction when monitoring the FLASH application running a 2-D Sod problem is shown in Figure 5. The event shown is that of cumulative *MPI_Allreduce()* time at each application iteration. This data uncovered an anomaly that caused the Allreduce performance to drop (probably due to external factors) during a single iteration at Step 100.

4.2 Histogramming Filter

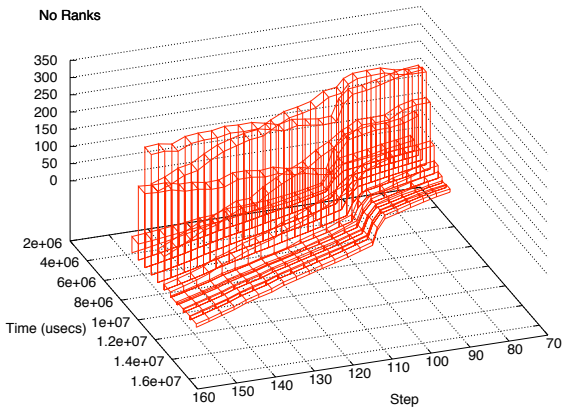

Figure 6. Allreduce Histogram

The *StatsFilter* while providing global summary statistics for every event, loses considerable spatial distribution information. A histogram is one method of reducing data while still maintaining a level of distribution information. The *HistFilter* extends the *StatsFilter* and provides histograms for each event in addition to the summary statistics. Given a number of bins, to perform histogramming accurately, the global min/max must be known (so that the ranges of the bins may be calculated). Below the root of the ToM tree, global information is not available. To be able to distribute the histogramming function across the intermediate nodes in the *ToM* tree, the global min/max first needs to be ascertained.

Figure 4 shows the *3-phase* approach used. Here in *Phase-A*, unlike for summary statistic calculation, i) it is sufficient to only determine the global min/max and ii) the *USF* continues to buffer the original performance data after *Phase-A*. Once the root of the tree (*Hist_FE FE*) receives the global min/max, it is packetized and pushed down the tree in *Phase-B*. On this reverse path downstream-filters (*DSF*) intercept this packet, retrieve the min/max and pass it on to the *USF* (whose memory address space they share as threads). In *Phase-C*, the *USF* at the lowest level first performs the binning of event data using appropriately sized bins. It pushes the resulting histograms up to its parent. In every round, the parent receives one histogram from each child, merges them and pushes upward again. The process repeats until a single histogram reaches the monitor at the root of the tree. Internal buffering within the filters ensures that the phases are non-blocking and can be pipelined. Figure 6 shows a portion of the histogram (from Step 70 to 150) corresponding to the summary statistic in Figure 5. The *HistFilter* was configured to use 10 bins to monitor 1024 MPI ranks with a *ToM* fanout of 8. The figure shows how the *Allreduce* time is unevenly distributed across the ranks and how that distribution evolves over time. The sudden increase in *Allreduce* time at Step 100 is seen here as well.

4.3 Functional Classification Filter

As an example, the spatial unevenness of the *Allreduce* across the ranks seen in Figure 6 may be attributable to network performance issues, load-imbalance issues or the existence of different classes of application ranks performing specific roles (i.e. not a purely SPMD model). In the latter two cases, it is important to distinguish between imbalance within the classes versus across them. The

ClassFilter groups the ranks into classes using a purely functional definition of a class. Given a performance profile, all of the event names are concatenated together and a hash of the resultant string is found. This is used as a *class-id*. Ranks with profiles that generate the same *class-id* are assumed to belong to the same functional class. Further, within these classes distributed histogramming using the 3-*Phase* approach is carried out. The output, then, is a set of histograms per event, one for each class. The method of classification can be application-specific or tailored to what the observer wishes. For instance, the *class-id* can be generated based only on a subset of application events (e.g. based on depth in the call-tree or if they are MPI routines). It must be noted that classification provides more information than simple histogramming. And it allows control of that detail through the *class-id* generation scheme. Hence, again, the type of classification must be traded-off against the extra data that it generates.

We use the Uintah Computational Framework (UCF) [6] for demonstration of varying functional classification schemes. The TAU profiling strategy for Uintah is to observe the performance of the framework at the level of patches, the unit of spatial domain partitioning. UCF is instrumented with events where the event name contains the AMR level and patch index. This case focuses on a 3 dimensional validation problem for a compressible CFD code. The domain decomposition in this case results in outer cubes that enclose eight (2x2x2) level 0 patches. Inner cubes cover the *interesting* portions of the domain that have been selected by the AMR subsystem for mesh refinement on level 1. Events are given names such as *"Patch 1 -> 1"* which represents the 2nd patch on level 1. The application is run over 64 ranks and monitored with a *ToM* fanout of 8 using the *ClassFilter* under different classification schemes. In the ***Default*** scheme all events in the profile are considered for creating the *class-id*. Only overall patch events are considered in ***Patch Only***. The ***AMR L1 Patch Only*** scheme goes a step further and restricts *class-id* calculation to level-1 AMR patches. Lastly, ***MPI Only*** looks at only the MPI events. In all the schemes, complete profile information from all events is still preserved.

Figure 7 plots the *no. of classes* (left y-axis) and the *reduction factor RF* (right y-axis) . The *RF* is the total non-reduced data size divided by data size with reduction. Both metrics are plotted for every application iteration,

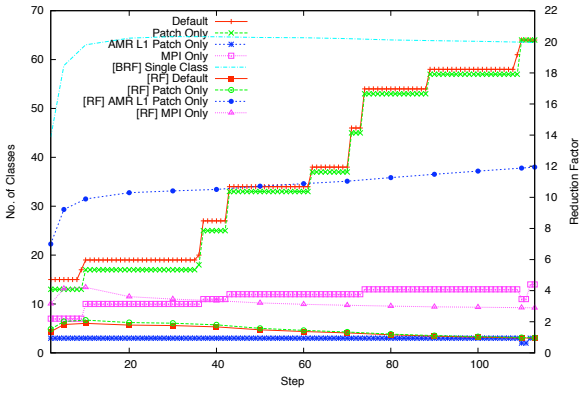

Figure 7. Functional Classification

with the latter being cumulative upto that iteration. The number of histogram bins is 5. The *base-reduction factor (BRF)* plots the reduction in profile data size achieved by performing histogramming without classification. The ***Default*** scheme eventually results in 64 unique classes (as the ranks diverge over time). Because there are as many unique patches created as available ranks, the ***Patch Only*** scheme behaves similarly. The *RF* in both the schemes reduces to 0.98 since there are as many classes as ranks and there is some overhead to histogramming. ***MPI Only*** results in 13 unique classes in the worst case with a resultant *RF* of 3. **AMR L1 Patch Only** results in an overall *RF* of 12 times and a maximum of 3 classes – class0: rank 0, class1: ranks that work on the first 8 large level 1 patches and class2: all other ranks. Larger number of ranks should increase the overall reduction factor.

5 Evaluation

5.1 Perturbation

	% Overhead		DUMP() (msec)	
Type	N=64	N=512	N=64	N=512
Tau-PM	0.049	0.23	-	-
ToM	0.56	0.77	3.49	3.60
ToM Reduce	0.17	0.70	3.29	3.55

Table 1. Perturbation Overheads

```
time = get_time();
for(i=0; i<iterations; i++) {
  work(usecs);
  TAU_DB_DUMP();
  MPI_Barrier();
}
time = get_time()-time;
tot.time = time-(usecs*iterations);
avg.time = tot.time/iterations;
```

Figure 8. Offload Benchmark

Any measurement scheme, in particular online monitoring, raises the issue of perturbation. The perturbation caused is due to overheads from both measurement and performance data offloads. Our real-world workload to evaluate perturbation is the FLASH3 [7] application running a 2-D Sod problem. The problem is scaled weakly from 64 processors to 512 processors. The different modes are: i) uninstrumented to acquire a baseline, ii) *TAU-PM*: with TAU instrumentation and data offload at termination, iii) *ToM*: with online data offload per iteration and iv) *ToM Reduce*: with online data offload per iteration along with histogramming. *ToM* was configured with a Fanout of 8. All experiments were run on the Atlas cluster at Lawrence Livermore National Laboratory. The mean (over 3 runs) of the % overhead over the baseline for the three cases are reported in Table 1. With over 120 application events (including all MPI events) in a real, complex parallel application the overheads in all cases were under 1%. For completeness the cost of performing the *TAU_DB_DUMP()* operation is also reported.

5.2 Data Reduction Costs

The different types of performance data reduction in *ToM* were demonstrated in Section 4. In each case extra work is performed in order to achieve the reduction (e.g. the *3-Phase* histogram). Under what circumstances is reduction beneficial, if at all? We evaluate the costs of performing that reduction versus the savings obtained from doing so. The metric used is the average time taken to perform a single offload at the *BE*. As the rate at which offloads occur increases beyond the service rate offered by *ToM* (and the underlying physical network), persistent queuing leads to buffer exhaustion and eventually to a blocked *send()* call. This cost is reflected in the average time to offload data onto *ToM*. While a non-blocking *send()* may not directly suffer these costs, the system will still require the same amount of time (or possibly more since offload rate will not be reduced by blocking) to eventually transfer the queued performance data. It should be noted that the experiments in this section are a severe stress-test of *ToM*.

We use a simple *offload benchmark* summarized in Figure 8. The `avg.time` variable is a measure of the mean of the worst offload time suffered per round across the ranks and is plotted as the *Benchmark Performance* in Figure 9. The x-axis (*Profile Period*) represents the interval between offloads in microseconds. The y-axis is the *ToM Fanout*. The **ToM** curve represents the the `avg.time` with no reduction and the **ToM Reduce** curve represents the case with reduction using histogramming.

In Figure 9(A) (application ranks, N=64), at relatively low offload rates both curves are overlaid. The knee observed in the curves is due to the offload rate increasing above the service rate. At *Fanout*=2, the knee in **ToM Reduce** occurs later than that in **ToM**. And at *Fanout*=4, while the knee occurs at the same rate, the magnitude of increase in **ToM Reduce** is smaller. In both cases, savings from data reduction clearly trump the costs of performing histogramming. At *Fanout*=8 **ToM Reduce** loses its advantage from data reduction. Reduction using histogramming has its own costs. For instance, it requires that each intermediate *ToM* rank has double the number of threads (due to the DownStream Filter). As *Fanout* increases the costs dominate the savings obtained from data reduction, suggesting that with low *N* and high *Fanout*, reduction does not help. In contrast, in Figure 9(B) where *N* is larger (256, 512), even with double the *Fanout* (16), **ToM Reduce** performs an order of magnitude better than **ToM**. As *N* increases, the savings obtained from reduction proportionally increases. Whereas the *Fanout* remains fixed and so too the cost of performing the reduction. This also suggests that at lower offload rates, much higher fanouts can be used, effectively reducing transport resource usage.

These experiments were run with a modest number of events (20). Repeating the runs with 50 and 150 events (results not shown here) had similar results. At small *N*, the cost of performing the reduction increased to be larger than the savings obtained. But with large *N*, the results closely resembled Figure 9(B), confirming that reduction

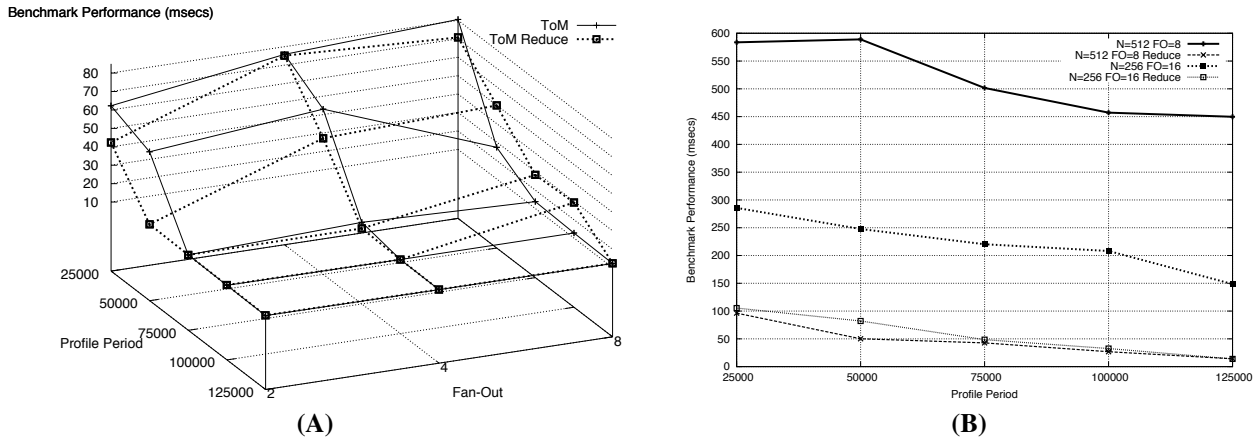

Figure 9. Offload Benchmark (A) N=64, (B) Large N (256, 512)

is beneficial with relatively large processor counts.

6 Conclusion

Our experiences with the *ToM* prototype confirm the high return on investment of additional system resources in support of performance monitoring. For instance, with a fanout of 64, overhead for additional transport nodes is just over 1.5% – a reasonable price to pay for the performance benefits. In addition to providing a scalable tree-structured network for consolidated data transfer, the ability to program MRNet for data analysis and reduction relieves the burden on front end processing. The *ToM* architecture and implementation provides a solid foundation for porting *ToM* to other platforms and evolving its capabilities in the future. Our immediate interest is to test *ToM* on extreme scale systems with tens of thousands of nodes. We will also develop new MRNet analysis components, especially ones that support feedback to the application on performance dynamics, such as for use in load balancing and resource (re-)allocation. In future, we envision connection of *ToM* to an interactive graphical monitor for real-time performance visualization and steering.

References

[1] "SDTPC: Workshop on Software Development Tools for Petascale Computing, Washington D.C," 1-2 August 2007, http://www.csm.ornl.gov/workshops/Petascale07/.

[2] S. Shende and A. D. Malony, "The TAU parallel performance system," *The International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–331, Summer 2006.

[3] A. Nataraj et al., "TAUoverSupermon : Low-Overhead Online Parallel Performance Monitoring," in *Europar'07: European Conference on Parallel Processing*, 2007.

[4] 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.

[5] J. Liu et al. , "Design and implementation of MPICH2 over InfiniBand with RDMA support," in *International Parallel and Distributed Processing Symposium (IPDPS 04)*, April 2004.

[6] J. Davison de St. Germain et al., "Uintah: A massively parallel problem solving environment," in *HPDC'00: International Symposium on High Performance Distributed Computing*, 2000, pp. 33–42.

[7] R. Rosner et. al., "Flash Code: Studying Astrophysical Thermonuclear Flashes," *Computing in Science and Engineering*, vol. 2, pp. 33–41, 2000.