# A Performance Monitor based on Virtual Global Time for Clusters of PCs

Michela Taufer[1,2], Thomas Stricker[2]

[1] Dept. of CSE
University of California, San Diego
taufer@cs.ucsd.edu

[2] Dept. of Computer Science
ETH Zurich, Switzerland
tomstr@computer.org

## ABSTRACT

*Debugging the performance of parallel and distributed systems remains a difficult task despite the widespread use of middleware packages for automatic distribution, communication and tasking in clusters. In this paper we present a performance monitoring tool for clusters of PCs that is based on the simple concept of accounting for resource usage and on the simple idea of mapping all performance related state of hardware performance counters and operating system variables backwards to the application level. In this way a monitoring tool can explain the most relevant performance metrics at a higher level that is easily understood by the application developer. The most important metric for distributed high performance applications remains the total execution time vs. the number of compute nodes involved, since it translates into the scalability of an application. As a detailed contribution of this paper, we closely look into what is needed to reverse map the low level performance counters at each node back through the middleware layer responsible for the parallelization and distribution. The specific problems encountered and dealt with are the creation of a flexible notion of global time for timestamping and the reassembling of performance data and an appropriate communication mechanism to minimize monitoring intrusion due to the additional networking traffic caused by the monitor. We show how our tool can be used to measure, explain and predict the performance and scalability of a distributed OLAP application running on clusters of PCs.*

**Keywords:** *Parallel and distributed systems, real-time performance analysis and evaluation, performance metrics, monitoring traffic, notion of time in distributed systems, clusters of PCs.*

## 1. INTRODUCTION

Up to this date a significant effort has been put into software systems for performance monitoring and performance debugging parallel and distributed applications on clusters of PCs. The systems include extensive monitoring and account for communication traffic for parallelization of loops and a variety of other instruments that are very helpful to a programmer who knows all the common methods to code a computation in parallel- or distributed way for higher speed. Most of the existing tools use some graphical means to display a wealth of information they collect about the running application. A brief survey is given at the end of the paper when describing related work. Despite all the research and implementation efforts, distributed systems still lack performance monitoring tools that are as simple to use and as clearly cut in their functionality as the common debuggers or profilers for uniprocessors are. This fact is recognized by the most important experts in the field as the primary hindrance to the success and widespread use of parallel and distributed computing [1].

As a general contribution to the state of the art we describe a novel performance monitoring tool that should be as simple as possible in its functionality and much easier to use as previous systems parallel or distributed applications. Most performance monitoring tools are heavily intrusive - either compile-time or at run-time. They require instrumentation at the source code level, at link level or do binary rewriting. In contrast our performance monitoring tool attempts to provide its instrumentation independently, side-by-side to any applicable middleware package. The tool attempts to avoid intrusion into the code whenever possible and works with the performance counters that are present in most microprocessors used in the compute nodes or with the performance information kept in the kernel of the common operation systems used in the compute nodes. The resulting performance monitoring information is gathered and assembled into a global performance assessment of the executing application.

As a specific contribution, we address four interesting issues related to the problem of collecting and reconstructing global performance monitoring data on distributed systems. First, we present an innovative way to deal with the additional network traffic due to the collection and the processing of performance monitoring samples. This keeps the intrusiveness related to addition communication traffic as low as possible. Second, we address and solve the problem of rebuilding a global notion of time within a parallel or distributed system using some accurately synchronized cycle counters to get a global time scale without requiring any additional synchronization messages. Third, we show how to reassemble the information of the hardware performance counters and the operating system performance variables into a global performance picture. Based on our previous experiences [2, 3, 4], we claim that in most applications there is a simple relationship between the machine resources required by the application and its execution time. The total execution time decomposes into parts that are largely determined by the usage of one single critical machine resource. Examples are part limited by the CPUs, the memory systems, the distributed disks or the communication system in a cluster of commodity PCs or a desktop grid. As we use a lossy communication mechanism to minimize intrusion while collecting monitoring samples, we compensate the lost samples in simple way by interpolation and by transmitting the totals of resource usage in each samples rather than just the increment. The final grand totals are transmit with a reliable request response protocol after the application completes and are guaranteed to arrive.

The resolution of our simple performance monitoring tool might

not be as high as some other more sophisticated tools, but the tool stays conceptually clean and helps to provide an easy way to analyze and predict the performance for general applications for which the user relies on an middleware for the parallelization and the distribution. We have used our performance monitoring tool on distributed databases running parallel OLAP workloads on a cluster of database PCs as well as on distributed protein folding calculations running on clusters of commodity PCs. While most functions of our software system are generic to any application some parts remain application and middleware independent. Towards the end of this paper we prove the viability of our performance monitoring tool as we use our tool in an experimental study to find the reason behind the insufficient scalability of a particular TPC-D query executing on a cluster of three or six PCs connected with different high speed networking technologies.
The rest of the paper presents the overall architecture and design principles of our tool, describes the implementation focusing on the solutions we devised for collecting and reassembling distributed performance data into a global picture and finally shows how our performance monitor can be used to measure, explain and predict the performance and scalability of a distributed OLAP application running on a cluster of PCs.

# 2. ARCHITECTURE OF OUR PERFORMANCE MONITOR

## 2.1 Resource Usage as Principal Metric of Performance

While the end user primarily looks at the total execution time of the application, the system architect is mostly concerned with the performance data related to the usage of the system resources like e.g. the number of floating point operations performed, the number of operands loaded from memory, the number of bytes communicated between the nodes.
Our performance monitoring system tries to link the two perspectives to each other by breaking the total execution time into partial execution times attributed to each type of resource, i.e. CPU, memory, disk and network. This simple model works well if the partial execution times of an application run can be cleanly attributed to one single critical resource at all times during the execution. Our experiences with different kind of applications from scientific computation to distributed database have shown that this is a viable first order approximation, than can explain main performance problem in practice [2, 3, 4].
This neat decomposition of execution time according to machine resource class allows the user and the systems architect to study the impact of each single resource to the system in an systematic way. To study interactions we can also apply the concept of factors and their confounding [5] to the study of resource usage.

## 2.2 Layered Software Systems

Most software systems running on clusters can be structured in three layers: the application code, the middleware packages supporting the application and distributing the computation and the operating system driving the hardware of the compute nodes.
Since we want our performance instrumentation to be compatible with all kinds of middleware packages and since we want to be as little intrusive as possible, we build our performance monitoring tool side-by-side to the middleware layers found in most modern distributed systems. We rely primarily on performance counters available in the microprocessor(s) and performance information available from the kernel of the operating system on the compute nodes.
The principal function of our performance monitoring tool is to map the detailed performance data available at the compute nodes into a global performance picture that can easily be related to the total execution time. The overall structure of the software system is shown in Figure 1.

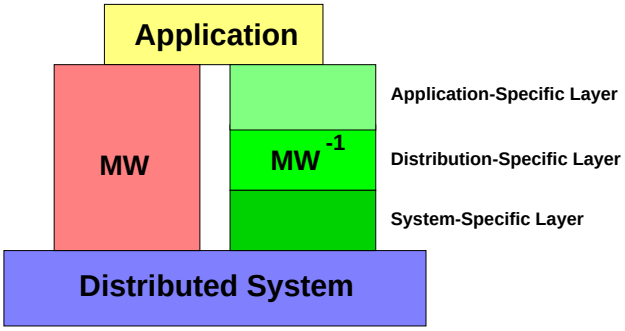


**Figure 1: Structure of a typical software system with our performance instrumentation.**

The functionality of the performance monitoring tool itself can also be partitioned into three layers.
The *system-specific layer* monitors and collects system-specific performance data. System-specific performance data includes information on resource usage and bottlenecks gathered over the timespan of the application-run. In our current prototype we monitor resources like the local CPUs usage, the memory usage, the local disks usage and the local network usage as sampled on each compute node. The monitoring mechanism is based on a dynamic sampling of performance counters (e.g. floating point operations, amount of traffic over the network or to and from the disks) already provided by most operating systems and microprocessors at regular intervals chosen by the user. The sampling takes place outside the kernel by daemon processes which use some performance hooks into an extended `/proc` file system and hardware performance counters provided by the microprocessors.
The *distribution-specific layer* collects the performance related data from several compute nodes and patches it into a single coherent view of the whole system to be handed over to the application-specific layer. The application traffic and the monitoring traffic are separated using different transport protocols over the network of the cluster of PCs or the desktop grid. Our performance monitoring tool inherits the master-slave setting from its middleware counterpart. The data locally collected by the system-specific layer is processed in the distribution-specific layer and sent to the monitoring master by a message passing mechanism. For performance reasons the monitoring master is kept on a separate node and collects all information from the other compute nodes. The information collected from the master is properly filtered by the slaves and made ready for processing at the application-specific layer. Our detailed study of the most important implementation issues for maintaining low monitoring intrusions with the inverted middleware is described in the next section.
The *application-specific layer* uses the global performance data of the entire system for application-level optimization and performance predictions. The topmost layer uses an analytical model of the specific application to map the response variables collected from the global view of the system into suitable suggestions for per-

formance tuning, e.g. changes of parameters that are performance relevant factors of the computation. The analytical model translates the elementary knowledge about the resource usage monitored at the system-specific layer and gathered by the distribution-specific layer into high-level answers on performance questions and bottlenecks suitable for suggesting optimizations to the user. More specifically the model comprises a set of formulas which allow the calculation of individual execution time contributions due to a single class of resources that is the bottleneck for this part of the application. For each application the model has to be adapted, validated and calibrated.
Most of the work described in this paper addresses the characteristic problems related to the distributed and parallel computation on clusters of PCs and therefore we tightly focus on the distribution layer. A detailed description of the whole system including all the layers is reported in the PhD thesis of the first author [6].

# 3. MECHANICS OF PERFORMANCE DATA COLLECTION

Because most high performance applications in our environment exploit task parallelism in a master-slave computation setting, our monitoring tool collects the performance data in a similar way by using the same master-slave setting. To ensure consistency and an accurate performance picture of the system and to capture the real-time behavior of the monitored application, some frequent and fast transmissions of performance data from the monitored slaves of the distributed system to the monitoring master are required. The subsequent sections present a method of performance data collection that is scalable and is minimizing monitoring intrusions. Particular attention is given to timestamping the samples with virtual barriers and minimizing the adverse effects of the monitoring data traffic by using UDP/IP communication and lossy communication in the monitoring layer.

## 3.1 Ordering the Monitoring Performance Data with Virtual Global Time

Monitoring tools in distributed systems must be able to patch the performance data of several nodes together into a consistent performance picture using global wall-clock time and a coherent notion of time across the many nodes of the distributed system. To solve this problem, a monitoring tool has to introduce some mechanisms of synchronization for sampling performance data. Mechanisms based on ordered events such as barrier messages for synchronizing the computation and the communication do always change the run-time behavior of the monitored application and introduce additional idle times changing the scheduling and execution of the processes.
To cope with the problem of maintaining a global notion of time while simultaneously reducing scheduling and execution intrusions, we propose to include a sophisticated notion of global clock based on accurately synchronized cycle counters in the microprocessors. We avoid unnecessary synchronization messages (introduced by mechanisms based on ordered events) through a precise synchronization at the start of the monitoring session. As the execution proceeds for a longer time interval we are relaxing the synchronization to a looser synchronization model as in [12]. The necessary synchronization for the timestamps of samples in our monitoring tools are given by looking at the highly accurate cycle counters in the CPUs of each participating node. Because of the UDP/IP protocol, the packets from the monitored slaves to the monitoring master are not guaranteed to arrive in the same order as they are sent - they are reordered based on their timestamps. Moreover, due to an unreliable transport we can no longer assume that that every packet is actually received at the master. Instead of enlisting TCP/IP we introduce timestamps and sequence numbers to account for what is received in an dedicated packet acknowledgment protocol. This gives us the option to drop samples in situations of heavy networking load. The cycle counter values together with the sequence number are assembled into a timestamp in all performance packets containing the sample information of a performance counter that is sent to the monitoring master.
Such timestamps act as virtual barriers for the monitoring master which is able to rebuild a global picture of the events of resource usage that occurred on the slaves using its own counters and the profiling information of the master. These timestamps of the packets received ensure a sufficiently accurate notion of global time similar to the work described in [13].

## 3.2 Reconstructing the Global Performance Count from Messages

Most of the monitoring overhead on the slaves is spent in the non-operational phase managed by timing routines. Most performance data is only available in the kernel, but most of the performance monitoring logic is kept external to the kernel and the data is obtained from the kernel through */proc file mechanism* in LINUX. The standard information available in the `/proc` file is augmented by additional performance counts to report the total number of blocks and the number of bytes exchanged from and to the local disks and the number of messages and bytes sent to or received from the network interface. As part of the development a kernel library has been implemented to permit a simple but efficient way to access the Performance Monitoring Counters (PMC) in the different Intel Pentium and Dec/Compaq/HP Alpha processors used in our projects [14]. The library offers a uniform interface under LINUX and Windows NT and supports the monitoring of symmetric multiprocessing (SMP) machines collaborating with the scheduler to account for used resources correctly. It works with the LINUX 2.2.x and 2.4.x kernels and has been validated against other performance monitoring mechanisms [15, 14].
The granularity of the sampling determines the size of the regular intervals by which the information about the local nodes is collected. Each sample comprises static performance data, like the node identifier and the clock rate of the CPU, and dynamic performance data. Among the dynamic performance data, our tool is able to monitor: (1) the CPU behavior in terms of total floating point operations and number of instructions computed, (2) the memory availability in terms of total, used and free memory on the node, (3) the disk performance in terms of number of read and write accesses as well as sequential and non sequential accesses, and (4) the amount of traffic transferred over the network interconnect in terms of amount total bytes received and sent as well as the total number of packets received and sent. The user sets up both the sample rate and the kind of performance counters when starting the monitoring. This process takes place by sending a start command to a controller daemon which runs on the same node of the monitoring master and lets the user to control the monitoring master. The user can also change the sampling rate at run-time, stop the monitoring of a single node or the whole pool of nodes by sending a proper command to the control daemon. Even when monitoring the whole range of listed counters, the information of each sample is written into packets whose size is less than 512 bytes. The total collection time for a packet is irrelevant: because of the coarse grain sampling, we have observed that the total execution time of a monitored application

and the total execution time of the same applications without the monitoring remains the same.
The monitoring master and its control daemon can either run on a dedicated node or on the same node on which the application master runs. In our setup, we leave the application user to decide where to put the application master and the monitoring master. Our monitoring master is responsible for putting together the local performance figures into a global view. The performance data that the monitoring master gets from each monitored slave are ordered sets of workload samples on the local nodes.
The notion of time that we use, i.e. the timestamps of the cycle counters, helps the monitoring master to maintain the concept of order for the samples. The concept of a confidence interval is used to measure the quality of the transmission. The less packets get lost, the larger is the number of samples. At the same time, the larger the number of samples is, the higher the confidence in the performance rebuilding will be.
We save the performance data collected by the monitoring master in simple log files/log streams rather than in a database (e.g. MySQL, Oracle). The efficiency of a file system or a simple socket increases the scalability of the monitoring tool in the capture phase under real time constraint at the cost of reduced flexibility and capability in the postprocessing phase that is not time critical. In [16] we question the use of databases for the simple storage needs of performance monitoring tools in and show that the overhead of a DBMS can significantly affect scalability of the tools, in particular the number of nodes which can be monitored by the master at the same time. The log files or log streams are passed to the application-specific monitoring daemon that reconstructs a global performance picture at an abstraction level most suitable to the application developer.
Each performance count related to a specific node is marked by the node identification and the CPU clock time as the information is picked up on the local node. The amount of data sampled locally at each node and sent as a performance montioring packet might be small by itself, but the total amount of information gathered from all the nodes grows rapidly in a large highly distributed system. At the same time, previous information can often be summarized for real-time performance evaluation later. Therefore we replace the initial postprocessing scripts by a concurrently executing daemon which processes and filters the performance data on the fly up to the application layer. In the application layer used for evaluating OLAP workload, the elaboration processing consists of a simple daemon which is constantly fed with the log files and simple sums the performance counters to provide the user with an application classification based on resource usage.
A more advanced application-specific layer that works side by side with more sophisticated distributed computing middleware can include a performance model of the application that does translate the elementary knowledge about the resource usage into higher level answers to relevant performance questions or for finding bottlenecks as starting points for performance optimization or for performance predictions on future systems configuration. A detailed description on how to take middleware and construct an apropriate performance data processing layer for each middleware layer in complex software system is reported in [6].

## 3.3 Example: Monitoring Communication Activity in OLAP

Most parallel and distributed computations have execution times that are much larger than lower bound given by the most critical resource (e.g. CPU cycles or bytes read from a disk). They also introduce additional intrinsic bottlenecks due to the process of parallelization or distribution of the work among multiple proecessors. An adequate performance monitoring tool must therefore be able to capture and evaluate a time dependent picture of resource usage and identify points in time when a particular resource is overused.

The critical machine resources in our simple example of a TPC-D query executed on a distributed database management system are the CPUs, the disks and the communication between the multiple worker nodes and the coordinator node of the distributed system. Figure 2 shows the amount of data communicated across the network over time. Due to the accurate notion of virtual global time (synchronized clock counters) we can identify two phases of the parallel computation. In the first phase (i.e. the first 200 seconds) there is no visible load on the communcation system. A a closer look the corresponding charts for CMU or disk usage usage would indicate the proper peak resource usage for the first phase. The communication load suddenly starts during the second phase of the computation. We can also see from the data rates that all the nodes communicate to the coordinator node simulaneously and that the total load on the coordinator is additive. The communication performance of the coordinator limits scalability. A closer look reveals some software inefficiencies in the communication protocols. In theory the bulk of the data should be uni-directional from the workers to the coordinator receiving all data to assemble the result of the distributed query - in practice a fine granular request/response protocol causes a large amount of unnecessary acknowledgement traffic slowing down the data transfer. Our performance tools readily suggest that it might be impossible to increase the communication system performance by simply adding faster network hardware.

## 3.4 Dealing with Intrusions due to Monitoring Traffic

The monitoring of communication performance in a distributed system immediately suggests that the performance of the application observed by monitors should not be affected by the monitoring process itself. The distributed systems community has proposed to remove monitoring intrusion effects after the run (i.e. a posteriori) by carefully restoring the original message order in a network and reenacting the application run. In the reenactment the monitoring messages are removed. This approach works only for distributed application with little communication and misses most subtle forms of congestion in the switches of the network or the adverse effect due to second order effects like e.g. slow down in the network interfaces due to limited bandwidth of IO busses. Other approaches attempt to reduce the monitoring intrusions with additional support in hardware- [7] and in software [8, 9]. The techniques which refer to the software approaches presented in [8, 9] apply some real-time estimate of the application computation time based on compensation for each intrusion-induced effect. Unfortunately, these kind of approaches do not properly suit to the monitoring of high performance distributed computing applications in clusters of commodity PCs well. A clean distributed systems approach, proposed in [10], tries to reduce the monitoring intrusions by re-engineering the underlying communication system (e.g. instrument the socket library), to restore the original order of messages and to remove the service communication entirely from the performance picture of the monitored application. While this last approach theoretically valuable, the approach is limited to token-ring systems or fully connected point-to-point networks.
We take a more practical viewpoint and solve the problem of data collection without re-engineering the transmission protocol of the application and without going into the complexity of complete re-

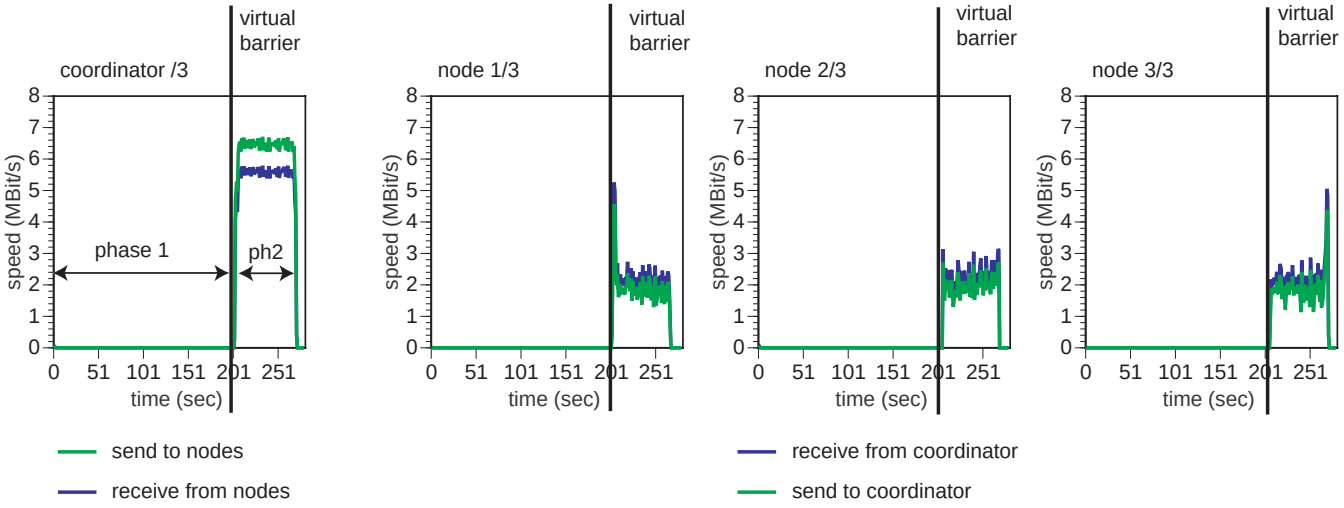


**Figure 2: Example of performance data collected by our monitoring infrastructure - the communication activity (number of bytes communicated per time unit) during the different phases of processing in a TPC-D Query 3 running on a distributed database management system on a cluster of PCs with a coordinator node (left) and three processor nodes (right).**

moval of monitoring intrusions a posteriori. Our approach is pragmatic and keeps intrusion at a minimum at the risk of a loss of precision in monitoring information. To keep the monitoring traffic of performance data as low as possible we use UDP/IP rather than TCP/IP for the performance monitoring traffic. Since UDP/IP does not attempt any re-transmission of lost messages, very little additional perturbation on the monitored application is introduced at the protocol level due to unwanted synchronization related to acknowledge messages between monitoring master and monitored slaves. It is assumed that in situation of heavy load due to the application (that uses TCP in most of the cases) the UDP traffic is simply lost and has little impact on the performance. Furthermore the performance monitors has knowledge about the amount of communication that is going on at a time and does scale back its transmissions accordingly. Moreover, because UDP has no constraints on the send rate of the performance monitoring packets and does not need to maintain connection states of the monitoring processes, this results in less overhead supporting a larger amount of monitored nodes than would be possible with TPC/IP [11].

### 3.5 Rebuilding Missing Performance Samples using Regression Models

Because our monitoring tool works on samples of performance information and not on whole populations [5] and these performance samples are frequently gathered and immediately integrated into a probabilistic framework provided by the application-specific layer, the loss of messages becomes a manageable problem for acquiring accurate performance views of the system.

Based on our experiments, we have observed that on clusters of PCs, the UDP/IP protocol works quite well and transmission errors are infrequent. Only when the network is over-utilized due to transmission bursts, some performance monitoring packets may get lost and need to be interpolated. Since our data is always expressed in absolute count (i.e. not deltas: samples dependent on previous data) there is only the possibility of missing occasional data samples along the time axis and not whole application runs as might happen when samples have a dependency on previous data. The missing samples can easily and reliably be reconstructed by interpolation. We indeed transmit total performance information rather than relative performance values into timestamped packets. This allow us to identify packets which went lost during the transmission and in the end allow us to rebuild the the global performance figure by using interpolations and message adaptive filters for specific performance counters (e.g. number of floating-point operations, number of accesses to the disks).

## 4. EXPERIMENTAL EVALUATION

In an experimental evaluation, we used our performance monitoring tool to analyze, explain and predict the performance of parallel and distributed applications which use the middleware functionality for the distribution of the computation on clusters of PCs. We gathered performance information in two different application domains, i.e. distributed molecular dynamics simulations [3, 4] and distributed OLAP database workloads. For brevity in this paper, we pick only the database application domain and execute the 17 queries used in the OLAP database benchmark TCP-D. The main performance problem of this class of OLAP applications on clusters of PCs becomes evident from the overall speedup picture shown in Figure 3.

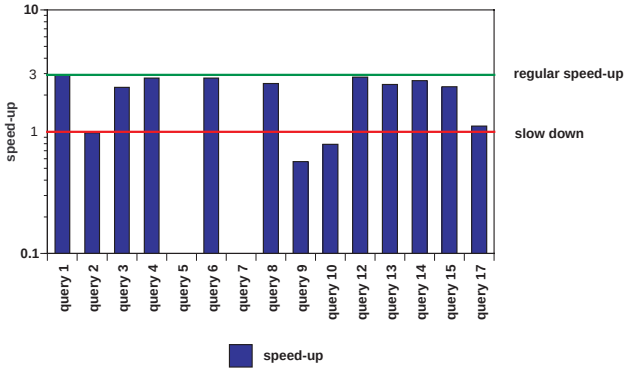


**Figure 3: Scalability for the TPC-D benchmark distributed across three nodes of a cluster of commodity PCs with TP-Lite.**

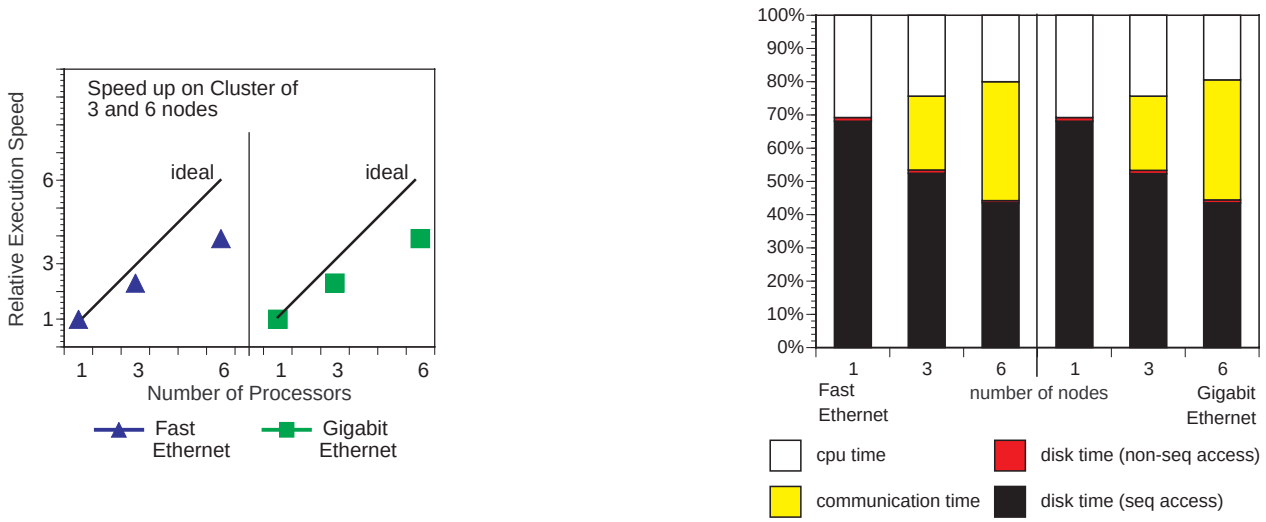


**Figure 4: Speed-up for a 1, 3 and 6 nodes cluster and corresponding relative usage time allocated to the specific machine resources for Query 3 (i.e. CPU, sequential access to the disk, non-sequential access to the disk and network usage). The data about the resource usage was collected and assembled in a global view by our performance monitoring tool.**

The chart shows unstable and unpredictable performance and speed-up numbers across the different queries of the benchmark workload. For our tests, we consider the distributed TPC-D benchmark on one, three and six PCs. The applications ran on a variety of different clusters of PCs within the CoPs cluster test-bed at ETH Zurich. The figure indicates some speed-up problems with as little as three nodes for some queries, while some queries get nearly optimal speed-up of almost three, others are undergoing a slow-down (i.e Query 2, Query 9 and Query 10) and some of them (i.e. Query 5 and Query 7) do not even complete in a reasonable time due to some inefficiency in parallel processing (or inefficiency in the software that is managing the parallelization). A complete empirical study of resource usage and scalability for all the 17 TPC-D queries, running with a distributed database system on clusters of PCs is published in [6, 2]. For the particular experiment with our performance monitor we execute Query 3 on a distributed database system built entirely from commodity hardware and software, involving a standard PC cluster running Linux and Oracle and PL/SQL at each node [17]. The distribution of the query is done automatically with TP-Lite, an experimental middleware tool provided to us by the database researchers at ETH [18]. Multiple instances of ORACLE run on several nodes of our 1 GHz cluster of PCs equipped with high performance SCSI disks and interconnected by Gigabit and Fast Ethernet interconnections. We can see that Query 3 is fairly efficient in distributed processing when running on three PCs with a speed-up 2.35 out of 3 maximal possible however, it loses ground on efficiency on six nodes with a speed-up of only 3.92 out of 6 - see Figure 4(left). Most surprisingly this speedups do not change when a ten times faster Gigabit Ethernet is substituted for Fast Ethernet.

The total size of the tables in the TPC-D database benchmark under investigation in this test is 10GB. Out of eight data tables of the TPC-D benchmark, we fully replicate the six smaller tables (i.e. Customer, Nation, Region, Supplier, Part, PartSupp) and we partition the two larger tables (i.e. Order and LineItem) equally among the nodes of the cluster. The amount of RAM is kept down to 256 MB to avoid unwanted caching of the two large relations of the database - in particular when they are partitioned.

Following our simple model of accounting for resource usage during the execution, we want to know from our tool how many CPU cycles are consumed for processing, how many bytes are read from and write to the disks and how much data is transferred as communication between the nodes. The application-specific layer of our monitoring tool assigns fractions of the total execution time to each resource used based on the pieces of performance information available from the performance counters of the microprocessor or the performance variables in the kernel which have previously been gathered in a global view by the distribution-specific layer described in the previous section.

Figure 4 (right) shows the total counts of relative resource usage as the faction of execution time allocated to the specific machine resources (i.e. CPU, sequential access to the disk, non sequential access to the disk and network usage) by our monitoring tool at the end of the query execution. As expected, there is no inter-node communication in the uniprocessor case (1 node) and the communication only becomes visible as we distribute the workload to multiple nodes (3 and 6 nodes) in the PC cluster. The distribution of the resources without the network is about 30% CPU and 70% disk and stays constant for larger clusters as well, implying almost linear scalability for the CPU and disk components. On the contrary the fraction of time spent in communication increases with larger machine and explains the lack of scalability.

Looking at just the total amount of data communicated and the theoretical peak capacity of the network, it came as an additional surprise that there is no improvement with the addition of Gigabit Ethernet which should theoretically be 10 times faster than Fast Ethernet.

Prompted by the sudden increase in communication time for a larger number of nodes, we ask our tool about the time variant information on communication loads. In addition to the total counts of resource usage, our performance tool is able to capture time and location of all resource usage by showing when, where and why these resources were exactly required. Because of the accurate notion of time introduced in our monitoring tool, we are able to detect the global time intervals when the various bottlenecks due to peak usage of the resources take place. We can take this high transient

loads into account by assigning additional execution time to the consumption of the resource responsible for the bottleneck.
For Query 3 the traces in Figure 5 gathered by our tool identify some burst network traffic and some imbalance between master/slave roles. There is also some high volume of unexpected service traffic that causes a high bi-directional volume despite the fact that application data is transferred only in one direction (from the nodes to the coordinator). In more detail, Figure 5 (left) shows the data rate of the whole communication activity gathered at the distribution-specific layer by the performance monitoring tool. An inquiry about the communication volume balance between the nodes indicates that the coordinator node shows increased activity while it is coordinating the processing of the query with three/six nodes (coordinator/3 and coordinator/6). At the same time the figure shows on the right low communication activity for the three/six individual processing nodes (node n/3 and node n/6). But in total the number of packets send and received are reasonably balanced.
To summarize, the graphs refute our initial suspicion that communication is highly asymmetrical due to the master/slave algorithm used in TP-Lite. The parallelizer tool is supposed to processes partial queries on each node and gathers the results in the coordinator at the end. In reality, the communication traffic is quite symmetrical and during the data transfer the coordinator sends almost as many bytes to request data as the nodes send for their answers. Furthermore, we observe that the peak communication rates on the coordinator node is approximately 6.4 MBit/s only, while the network that can sustain one GBit/s (i.e. 160 times more) under good conditions. The performance monitoring tool correctly points out some severe efficiency problems in the communication mechanism of the Oracle DBMS used in this experiment, leading to a communication limited scalability of Query 3 for no good hardware reason. Our Gigabit interconnected 1 GHz PC cluster is a fairly balanced system (by PC Cluster standards) and should therefore be able to execute such workloads much better than they did in the software configuration under investigation.
Database experts suggested to us to study the reason for such a suboptimal performance by looking at Oracle's built-in performance monitoring and tuning tools [19, 20]. Unlike our tool, such middleware specific hooks for performance analysis work on the basis of data collection inside the DBMS and not by mapping the microprocessor's or operating system's performance counts back to the application level. The Oracle built-in performance traces did not reveal much interesting news to explain the bad communication performance and they were only helpful to remove one misguided optimization of the automatic query optimizer. Furthermore the data from the DBMS performance tool cannot be easily related to the technical data of the platform, like e.g. the raw read performance of the SCSI disks or the raw throughput of communication links as our monitoring tool does.

## 5. RELATED WORK

Most of the existing monitoring tools come in the form of tool-kits or subroutine libraries. The prevalent principle is *code reuse* [21] instead of design reuse. On the other hand, our monitoring tool has an overall simple systems structure and comprises parts of the design and code which is reused independently of the application monitored or performance metrics investigated. It emphasizes *design reuse* over code reuse since its customization to a new application class or a new platform might well require a little bit of coding within the tool. e.g. moving to a different CPUs with different performance counters.
In the context of the *performance analysis of scientific computation applications*, most of the current methods for performance analysis require that the decisions related to instrumentation are taken at the beginning of a monitoring session and remain fixed during the execution of the monitored application. Changes of the performance metrics to be monitored requires a re-engineering or at least a re-linking of the application binary or even a redefinition and rewriting of the performance monitoring. Our tools allows the analysis of data during the collection phase and during the execution of the monitored application. The user can also redefine the data collected or the interval between two performance samples by dynamically acting on the parameters or the granularity of the data collection. Tool-kits like SvPablo [22, 23] capture performance data on platform architectures using C, Fortran and HPF compilers to instrument the source code and later visualize performance information. SvPablo requires the generation of a new version of the source code containing links to the selected events to be monitored. Our approach accomplishes the monitoring goals without any intrusion into the application code. Paradyn [24] is a quite sophisticated monitoring tool which requites that performance instrumentation is inserted into the application program and modified during execution. The strengths of our approach compared to Paradyn is the simplicity of the instrumentation and the and the dynamic behavior of the monitoring process. Using tools like VAMPIR [25, 26] and dyninstAPI [27] programs require re-linking with a specialized library. For our monitoring tools, no re-linking of the application is required. Moreover, VAMPIR keeps the performance data locally in each processor's memory saving them to disk when the application is about to finish for post-mortem optimization. On the contrary, our tools allows a dynamic processing of the information at runtime: during the application execution the trace data is sent in small, non-intrusive packets to a monitoring master. The Autopilot monitoring tool [28] provides a method for modeling and predicting using performance contracts between the application and the system. Autopilot provides read access to the remote application hosts via software sensors embedded in the application code and therefore requires a modification of the code before the application is executed. Contrary to the method proposed in [28], we do not need to insert any embedded components into the code of the application under investigation before the application is executed. We also do not specify any a priory expectation of the performance to be observed. Tools like $W^3$ Search Method [29] try not to embrace the fixed approach to data collection present in the tools listed above. Similar to our monitoring tool, the $W^3$ Search Method is application and machine independent. In addition, our monitoring tool is middleware independent.
In the context of the *performance analysis of high performance distributed database applications*, most DBMS packages (e.g. ORACLE [19]) incorporate elaborate instrumentation for performance monitoring and performance tuning, but such instrumentation normally works on the basis of data collection within the DBMS and do not directly relate to the usage of physical resources in distributed systems as our monitoring tool does.

## 6. CONCLUSION

Performance analysis in parallel and distributed computing systems like clusters of commodity PCs remains a highly difficult task, since we are still lacking simple tools that are capable of providing the user with performance insight without requiring either a lot of knowledge about distributed programming or a significant effort for instrumenting the middleware and application source codes with additional performance monitoring hooks.
As a conceptual contribution to this area of research, we present a

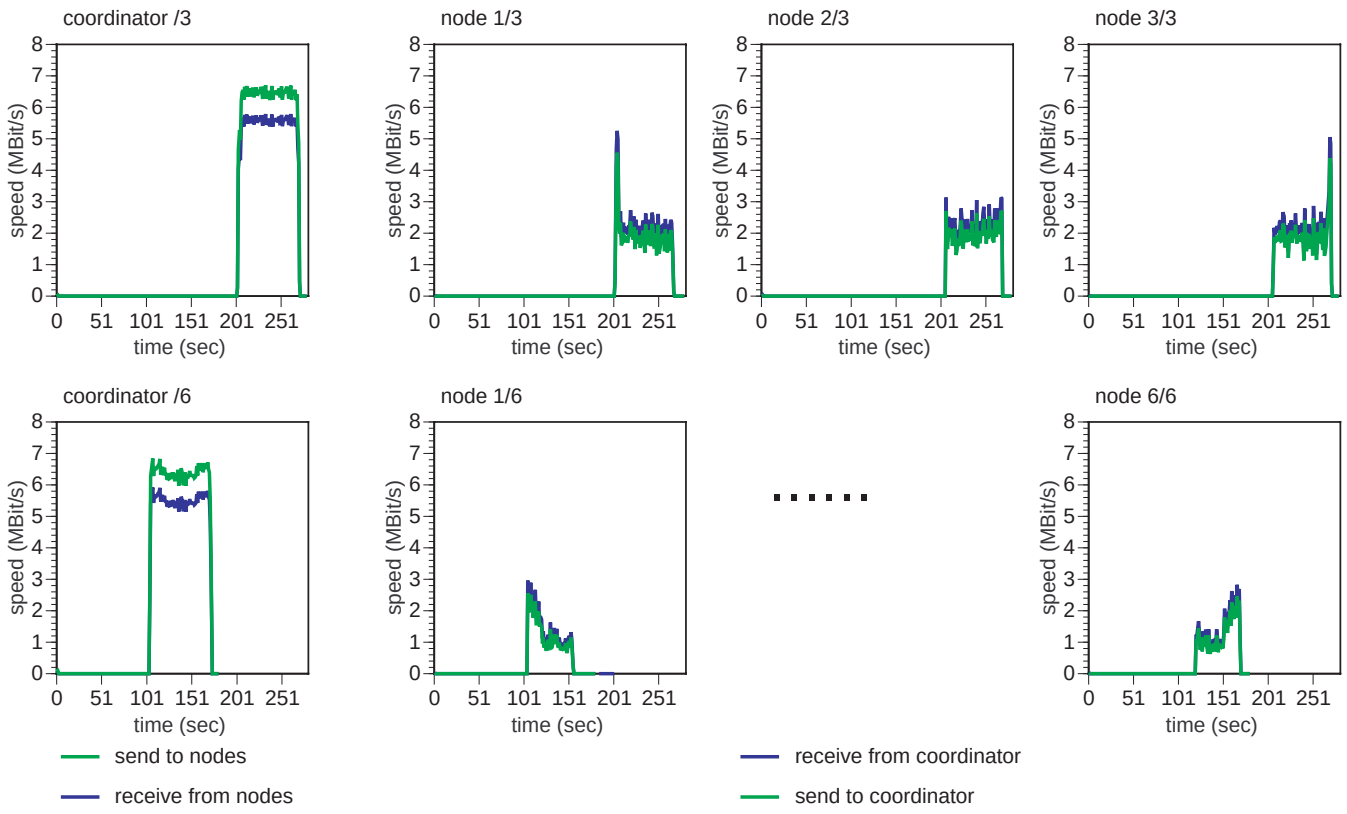


**Figure 5: Communication activity during the the processing of Query 3 for three/six nodes on the coordinator node (left) and on each of the three/six processor nodes (right).**

novel, general structure of a simple, but efficient performance monitoring tool for parallel and distributed applications. Our monitoring tool is built on one simple functionality and is therefore easy to understand. Unlike many sophisticated performance tools, our monitoring tool does not require re-engineering of the application nor the middleware but - instead - maps the performance counters of the microprocessors and performance variable available at the operating system level backwards to some high level representation that is more suitable for the proper analysis by the application writer. The underlying performance model is based on the simple concept of decomposing the total execution time of a distributed application into partial execution times that are broken down according to a vector of resource usage.

As an implementation contribution, we identify and solve several interesting implementation issues related to the collection of performance data on a Clusters of PCs and show how a performance monitoring tool can efficiently deal with all incurring problems. First, we show how our tool is able to capture inefficient execution due to resource bottlenecks with a low monitoring intrusion just by looking at the traces of resource usage all over the entire system in a fairly loose notion of time. We use timestamps along with all counter samples to correlate and assemble the performance data into a global performance log at the performance monitoring master. Sampling all performance monitoring information at a high rate is limited by the adverse effects of monitoring intrusion. Additional network traffic generated by the data collection must be bounded and kept at a minimum. We solve this problem by relying on a loosely collection scheme, that allows for dropping samples at in some time steps of intense communication in the application. Since any additional flow control of the monitoring traffic can adversely affect a running behavior of the distributed application, we use unacknowledged UDP/IP transfers instead of reliable TCP/IP for monitoring - at the risk of loosing samples. To cope with the possible loss of performance data, our tool rebuilds missing performance samples using regression models and maintains accurate total counts for all resource usage.

As an experimental contribution, we demonstrate the utility and the correct mode of operation of our performance monitoring tool for Query 3 - one of the least scalable and most interesting queries among the 17 queries in the TCP-D benchmark. The execution of the benchmark is distributed over three or six nodes of a cluster of commodity PCs and two different networking infrastructures are studied. Our performance monitoring tool can easily identify that Query 3 is a representative of a set of *network-limited* queries. This analysis is obtained without requiring re-engineering of or excessive intrusion into either the DBMS or the algorithm for distribution. The successful use of our tool with a SQL query distribution middleware and some replicated ORACLE DBMSes at each node shows that a performance monitor can be built and used even in the presence of middleware as a black-box that can neither be recompiled nor instrumented in another way.

Due to its simple mode of operation and its accurate notion of time, our tool further identified and explained the extremely bad communication efficiency due burstyness of the communication activity

and the inefficient protocols used by the Oracle DBMS for communication within a cluster. The highly similar runs with and without Gigabit interconnect hardware clearly point to software inefficiency behind limited scalability of Query 3 and demonstrate that a better communication system in the PC cluster would not have helped.

## 7. REFERENCES


[1] ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'03). Panel: Future architectures and programming models for high performance computing. http://ppopp.lcs.mit.edu/, 2003.

[2] M. Taufer, T. Stricker, and R. Weber. Scalability and Resource Usage of an OLAP Benchmark on Clusters of PCs. In *Proc. of the Symposium of Parallel Algorithms and Architectures, ACM SPAA '02*, Winnipeg, Manitoba, Kanada, Aug 2002.

[3] M. Taufer, E. Perathoner, A. Cavalli, A. Caflish, and T. Stricker. Performance Characterization of a Molecular Dynamics Code on PC Clusters - Is there any easy parallelism in CHARMM? In *Proc. of IPDPS 2002, IEEE/ACM International Parallel and Distributed Processing Symposium*, Fort Lauterdale, Florida, Apr 2002.

[4] M. Taufer, T. Stricker, G. Roos, and P. Guentert. On the Migration of the Scientific Code Dyana from SMPs to Clusters of PCs and on to the Grid. In *Proc. of the CCGRID 2002, IEEE International Symposium on Cluster Computing and the Grid*, Berlin, Germany, May 2002.

[5] R. Jain. *The Art of Computer System Performance Analysis*. Wiley Professional Computing, 1991.

[6] M. Taufer. *Inverting Middleware: Performance Analysis of Layered Application Codes in High Performance Distributed Computing*. PhD thesis, Laboratory for Computer Systems, Department of Computer Science, Swiss Federal Institute of Technology (ETH) Zurich, Oct 2002.

[7] J. Jeffrey, P. Tsai, K.Y. Fang, and H.Y. Chen. A Noninvasive Architecture to Monitor Real-Time Distributed Systems. *Computer*, 23(3):11–23, Mar 1990.

[8] A. D. Malony and D. A. Reed. Models for Performance Perturbation Analysis. In *Proceedings of 1991 ACM/ONR Workshop on Parallel and Distributed Debugging, ACM SIGPLAN Notices, volume 26, number 12*, pages 15–25, Santa Cruz, CA, Dec 1991.

[9] J.A. Gannon, K.J. Williams, M.S. Andersland, J.E. Lummp, Jr., and T.L. Casavant. Using Perturbation Tracking to Compensate for intrusion in Message-Passing Systems. In *Proceedings of the 14th International Conference on Distributed Computing Systems*, pages 414–423, Los Alamitos, CA, USA, Jun 1994. IEEE Computer Society Press.

[10] W. Wu, R. Gupta, and M. Spezialetti. Experimental Evaluation of Online Techniques for Removing Monitoring Intrusion. In *Proceedings of the 2nd SIGMETRIC Symposium on Parallel and Distributed Tools*, pages 30–39, Oregon, USA, Aug 1998.

[11] J. Postel. User Datagram Protocol. Technical Report RFC 768, ISI, Aug 1980.

[12] G.C. Fox. What have we learnt from using real parallel machines to solve real problems? In *Proceedings of the 3rd conference on Hypercube concurrent computers and applications*, pages 897–955, Pasadena, CA, USA, Jan 1988.

[13] L. Lamport. Time, Clocks, and the Ordering of Events in Distributed System. *Computer*, 21(7), Jul 1978.

[14] P. Stähli. Extention of a Performance Counter Library for Pentium Series. Technical report, 2001.

[15] R. Brand. Pentium II Performance Counting Library. Technical report, Swiss Federal Institute of Technology, Zurich, Switzerland, 1999.

[16] P. Cicotti, M. Taufer, and A. Chien. DGMonitor: a Performance Monitoring Tool for Sandbox-based Desktop Grid Platforms. In *Submitted to the Fourth International Workshop on Software and Performance (WOSP 2004)*, San Francisco, CA, Jan 2004.

[17] Oracle Corporation. Oracle8. http://www.oracle.com/, 1997.

[18] K. Böhm, T. Grabs, U. Röhm, and H.-J. Schek. Evaluating the Coordination Overhead of Synchronous Replica Maintenance in a Cluster of Databases. In *Proc. of the 6th International Euro-Par Conference*, pages 435–444, Munich, Germany, August 2000.

[19] M. Gurry. *Oracle SQL Tuning Pocket Reference*. O'Reilly, 2001.

[20] C. Dye. *Oracle Distirbuted Systems*. 1st Edition. O'Reilly, 1999.

[21] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley Professional Computing Series, 1995.

[22] L.A. De Rose and D.A. Reed. SvPablo: A Multi-Language Architecture-Independent Performance Analysis System. In *Proc. of the 1999 International Conference on Parallel Processing (ICPP'99)*, pages 311–318, Aizu-Wakamatsu, Japan, Sep 1999.

[23] D.A. Reed, R.A. Aydt, R.J. Noe, P.C. Roth, K.A. Shields, B. Schwartz, and L.F. Tavera. Scalable Performance Analysis: The Pablo Performance Analysis Environment. In *Proc. of the Scalable Parallel Libraries Conference (SPLC'93)*, October 1993.

[24] B.P. Miller, M.D. Callaghan, J.M. Cargille, J.K. Hollingsworth, R. B. Irvin, K.L. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn Parallel Performance Measurement Tools. *Computer*, 28(11):37–46, Nov 1995.

[25] W. E. Nagel, A. Arnold, M. Weber, H.C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and Analysis of MPI Resources. *Supercomputer*, 12(1):69–80, 1996.

[26] H. Brunst, H.-Ch. Hoppe, W.E. Nagel, and M. Winkler. Performance Optimization for Large Scale Computing: The Scalable VAMPIR Approach . In *Proc. of ICCS2001*, San Francisco, California, May 2001.

[27] B. Buck and J.K. Hollingsworth. An api for runtime code patching. *The International Journal of High Performance Computing Applications*, 2000.

[28] F. Vraalsen, R.A. Aydt, C.L. Mendes, and D.A. Reed. Performance Contracts: Predicting and Monitoring Grid Application Behavior. In *Proc. of the 2nd International Workshop on Grid Computing/LNCS (GRID 2001)*, Denver, Colorado, Nov 2001.

[29] J.K. Hollingsworth and P.J. Keleher. Prediction and Adaptation in Active Harmony. *Cluster Computing*, 2:195–205, 1999.