Journal of Information Assurance and Security 1 (2006) 59–77

Integrating Intelligent Anomaly Detection Agents into Distributed Monitoring Systems

German Florez-Larrahondo, Zhen Liu, Yoginder S. Dandass, Susan M. Bridges, and Rayford Vaughn

Department of Computer Science and Engineering
Mississippi State University
Mississippi State, MS 39762
{*gf24,zliu,yogi,bridges,vaughn*}*@cse.msstate.edu*

Abstract: High-performance computing clusters have become critical computing resources in many sensitive and/or economically important areas. Anomalies in such systems can be caused by activities such as user misbehavior, intrusions, corrupted data, deadlocks, and failure of cluster components. Effective detection of these anomalies has become a high priority because of the need to guarantee security, privacy and reliability. This paper describes the integration of intelligent anomaly agents and traditional monitoring systems for high-performance distributed systems. The intelligent agents presented in this study employ machine learning techniques to develop profiles of normal behavior as seen in sequences of operating system calls (kernel-level monitoring) and function calls (user-level monitoring) generated by an application. The Ganglia monitoring system was used as a test bed for integration case studies. Mechanisms provided by Ganglia make it relatively easy to integrate anomaly detection systems and to visualize the output of the agents. The results provided demonstrate that the integrated intelligent agents can detect the execution of unauthorized applications and network faults that are not obvious in the standard output of traditional monitoring systems. Hidden Markov models working in user space and neural network models working in kernel space are shown to be effective. Simultaneous monitoring in both user space and kernel space is also demonstrated.

Keywords: anomaly detection, performance monitoring, hidden Markov models, artificial neural networks.

1 Introduction

In April 2004, several academic supercomputing laboratories were targeted by "unsophisticated" attackers who logged into the system using compromised accounts. After the incident was noticed (in some cases several weeks after the intrusions had taken place) at least one supercomputing center reported that the attackers were executing distributed versions of well-known password cracking tools such as "John the Ripper," using standard MPI (Message Passing Interface) applications [1]. Even if the latest operating system patches are installed, the best commercial and open source firewalls are in place, and traditional intrusion detection systems are deployed, this type of unauthorized action can be concealed. Additionally, previous research by the authors has demonstrated several specific attack techniques against high-performance clusters. These attacks, while not yet reported in general use, have been implemented in research environments and shown to perturb normal operations in a cluster of Linux workstations [2–4]. Application-specific anomaly detection systems hold the promise of detecting such attacks.

These are a few examples of the lack of protection in sensitive distributed systems where a wide variety of internetworked computational resources are combined. Often, in order to contain costs, commercial off-the shelf (COTS) components are used and these typically provide sparse monitoring mechanisms. Furthermore, despite careful monitoring of the software development process, flaws in the design and implementation of COTS systems create opportunities for unexpected system failures, user misbehavior and computer attacks.

In the not-too-distant future, it is likely that high-performance clusters will be employed in safety critical environments such as the dashboards of automobiles where they will facilitate drive-by-wire applications, in the nose-cones of anti-missile missiles where they will control targeting maneuvers, and in the cockpits of high-performance aircraft where they will handle various avionics applications [3]. Typically, such systems will mostly consist of a static suite of software applications exhibiting regular behavior that can be characterized and monitored. Furthermore, corrective, perfective, and adaptive maintenance will be performed on the software over time, increasing the risk of introducing malicious exploits. Detecting anomalies and reporting them quickly will be an essential requirement in these systems.

The Center for Computer Security Research at Mississippi State University *(http://www.cse.msstate.edu/~security)* has been working on the problem of anomaly detection in high-performance computer environments by using machine learning techniques to build intelligent anomaly detection agents

Received December 27, 2005

1554-1010 $03.50 © Dynamic Publishers, Inc.

[2–6]. Lightweight detection systems with high accuracy have been successfully implemented, but several issues related to scalability and usability remain. In previous work with agent-based detectors in high-performance clusters of workstations, the authors used a centralized collection scheme in order to gather the results of the anomaly detection systems running on each node of the distributed system. The central host displays the status of the cluster system. However, when the number of nodes in the cluster is large, communication with this central host becomes a bottleneck. Therefore, alternative scalable solutions are required. Furthermore, determining how to meaningfully convey the voluminous output from the various distributed anomaly detectors to the system administrator is an additional challenge.

Several distributed cluster monitoring systems are available in the high-performance computing community that display cluster status and performance information to system administrators. Therefore, a natural extension of previous research is the integration of intelligent anomaly detection agents (IADAs) into traditional cluster monitoring systems. This paper describes the integration of two IADAs into the Ganglia distributed monitoring system (developed by Massie, Chun and Culler at the University of California, Berkeley [7]). Ganglia is open-source, can be compiled for a wide variety of Unix/Linux flavors, provides detailed documentation, and has been used successfully by more than 500 cluster and grid installations around the world. Furthermore, the Ganglia PHP-based web front end provides an effective mechanism for visualizing the overall status of the distributed system. This display mechanism can be leveraged for displaying the behavior of parallel applications as reported by the IADAs.

2 Related Work

Verifying a program's behavior by analyzing the processes, methods, tasks, and function calls that a program executes has been an active field of research. Forrest and Longstaff [8] authored one of the first research papers on the analysis of system calls. An extended overview of their work is described in [9]. Other anomaly detection algorithms include the EMERALD system [10], Somayaji's pH [11] and Eschenauer's ImSafe [12]. Warrender, Forrest and Pearlmutter [13] performed a comparison of different algorithms that performed anomaly detection of privileged UNIX programs using system calls. In their previous work, the authors have successfully applied different artificial neural networks and boosting algorithms for detecting anomalies in well-known sequential UNIX applications and in parallel applications executing on high-performance clusters of workstations [14, 15].

Markov processes are widely used to model systems in terms of state transitions. Some detection algorithms that exploit the Markov property implement hidden Markov models (HMM), Markov chains, and sparse Markov trees. Lane [16] used HMMs to profile user identities for the anomaly detection task. An open problem with this profiling technique is the selection of appropriate and effective model parameters. Other experiments performed by Warrender, Forrest, and Pearlmutter [13] compared an HMM with algorithms such as *s-tide* and RIPPER for anomaly detection using sequences of system calls. They concluded that the hidden Markov model exhibited the best performance but was also the most computationally expensive of all the models considered.

A number of applications have been implemented to gather data from the execution of parallel programs in a cluster of workstations, but none were developed with anomaly detection as an objective. Some examples include the automatic counter profiler [17] and the Umpire manager [18].

Several monitoring tools have been published and used successfully in real-world distributed environments. As an example, PerfMC, the monitoring system created by Marzolla [19], implements a performance monitoring systems for a large computing cluster. Some of the metrics collected by PerfMC are available space on */tmp*, */var* and */usr*, cached memory, available free memory and total swap, among others. Lyu and Mendiratta [20] use resource monitoring for reliability analysis of a cluster architecture in their RCC (Reliable Clustered Computing) system and present a Markov reliability model for software fault tolerance. Augerat, Martin and Stein [21] implement a scalable monitoring tool that uses *bwatch*, a Tcl/Tk script designed to watch the load and memory usage of Beowulf clusters.

Parmon, another example of a cluster-monitoring tool for Beowulf clusters, was designed and implemented by Buyya [22]. It monitors system resources utilization, process activities, system log activities, and selected kernel activities for each node in a distributed system. This performance monitoring system is based on a client-server model in which nodes to be monitored act as servers and monitoring nodes act as clients. Performance information is collected and stored in a *Parmon-server* and a GUI based client is responsible for data visualization.

Finally, Supermon [23] a research project conducted at Los Alamos National Laboratory, consists of three distinct components: a loadable kernel module providing performance information about each node, a single-host data server (*mon*) and a data concentrator (*Supermon*) that summarizes performance information from many nodes into a single data sample. Supermon uses a data exchange protocol consisting of data in *s-expressions* format introduced originally as part of the LISP programming language. The simple and recursive form of *s-expressions* allows Supermon to encode arbitrarily complex data. Unlike Parmon, Supermon provides an architecture to collect and represent performance information from a distributed system, but it does not provide a graphical user interface for visualization of the overall status of the system.

3 Ganglia: A Distributed Monitoring System

The authors selected Ganglia as the test platform for integrating IADAs into a distributed monitoring system because Ganglia is open-source, is supported on many commonly used platforms, provides for easy integration of new components for monitoring and visualization, and is widely used in the cluster computing community [7].

Ganglia was developed by Massie, Chun, and Culler at the University of California, Berkeley [7] as a scalable distributed monitoring system for high-performance computing systems. This system is able to collect between 28 and 37 different built-in and user-defined metrics "which capture arbitrary application-specific state" [7]. Ganglia is based on a hierarchical design targeted at federations of clusters, where a multicast-based listen/announce protocol is used to monitor state within clusters and a hierarchy tree of point-to-

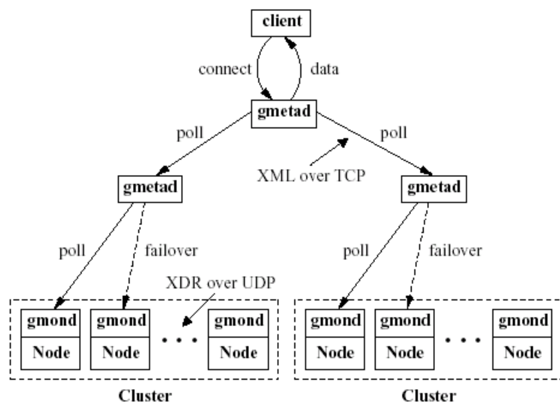

Figure 1: The Ganglia architecture (Taken from Massie, Chun and Culler [7])

point connections among representative cluster nodes is used to federate clusters and aggregate their state. Membership is maintained by using the reception of a heartbeat from a node signaling the node's availability. Each node monitors its local resources and sends multicast packets containing monitoring data to a well-known multicast address whenever significant updates occur. Thus, all nodes always have an approximate view of the entire cluster's state, and this state is easily reconstructed after crash recovery.

Ganglia uses the RRDtool (Round Robin Database) to store and visualize historical monitoring information, XML (Extensible Markup Language) for data representation, and XDR (External Data Representation standard) for compact data transport.

Figure 1 (taken from [7]) shows the basic software architecture of Ganglia. Ganglia uses two daemons, *gmond* and *gmetad*. The former monitors single nodes, whereas the latter integrates data from multiple clusters. Because the experiments reported in this paper were conducted on a single Linux cluster, the set of Ganglia metrics reported in this paper were collected from *gmond*.

4 Anomaly Detection

An anomaly detection system learns a *baseline-model* from a set of observations of a computer system operating under normal conditions and recognizes unusual and potentially dangerous events in the system. Any deviation from the set of normal patterns is considered an anomalous event. This work assumes an environment where the application base is well defined, but the user-base may not be. Therefore, patterns of usage are expected to emerge that can be used to detect irregularities in the execution of parallel programs. These irregularities include user misbehavior, intrusions, corrupted data, deadlocks, and failure of cluster components.

Current monitoring systems such as Ganglia are able to detect some of these irregularities by testing individual services with simple message exchanges among the cluster components. Other configuration management systems provide methods for configuration and for prevention of errors in a cluster. However, no single system can find all possible errors, misconfigurations, and runtime irregularities. Furthermore, traditional monitoring systems typically cannot provide

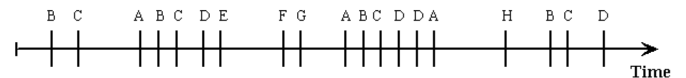

Figure 2: A sequence of events produced over time

useful information about the status of specific distributed applications because they show the overall status of the whole system.

An application-specific anomaly detection system can provide a qualitative measure of the execution of a job that answers the question *"is the parallel program being executed correctly?"* Detection of anomalous program execution can also play an important role in providing the inner layers of protection in defense-in-depth security architectures. For example, many cluster computer installations typically enforce tight authorization and authentication policies at the *head* node while leaving the compute nodes relatively less secure. This is because only the head node is accesible from outside the cluster; the compute nodes are not directly connected to an external network. However, this security configuration renders the cluster vulnerable to misuse by insiders and by outsiders who have compromised the passwords of legitimate users. Additionally, application-specific anomaly detection can also be used to enhance the security of systems in which *classes* of users, as opposed to individual users, are permitted to submit jobs. In such systems, profiling behaviour of users may be of limited use for anomaly detection.

Anomaly detection generally consists of the following three phases:

1. Samples of sequences of normal events in the system are collected, generating a set of observations. The discrete events generated by the application being monitored include operating system calls (low level information, known as kernel-level monitoring) and/or application library function calls (high-level information, known as user-level monitoring).
2. Machine learning algorithms are applied to the dataset collected in the previous step. In the experiments conducted for this paper, the profiles were acquired using artificial neural networks and Hidden Markov models. In each node of the cluster, the *profile* of the application is used as the input to an IADA.
3. New sequences of events in the system are collected, and the intelligent anomaly detection agents determine in real-time if the sequence of events is sufficiently similar to the *profile*. If the application is not behaving as expected, a counter of anomalies is increased and reported to the Ganglia database.

5 Intelligent Anomaly Detection Agents

A problem of interest in many domains is the characterization of signals produced by a real-world process in terms of simple models. Such models can provide an abstract description of the process and can be embedded in practical systems to aid in complex decision problems [24].

This research deals specifically with discrete-valued signals, where the stream of events being monitored is categorical, (*i.e.* it can be divided into finite groups). Examples of categorical variables include DNA bases, operating system (OS)

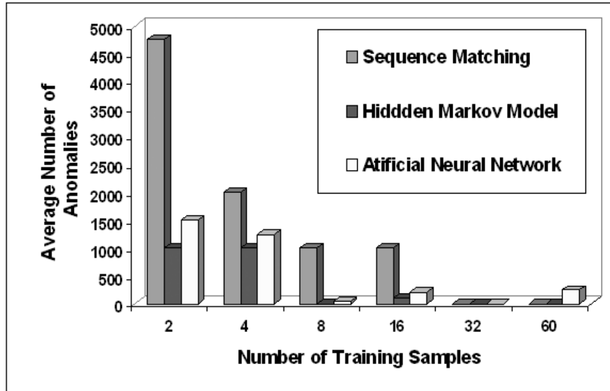

Figure 3: Average number of false positives detected in 60 test instances of FFT

commands typed by a user on a console, computer security audit events, and network requests among many others.

Previous research in the area of anomaly detection in complex software applications [3, 4] has demonstrated that artificial neural networks (ANNs) and hidden Markov models (HMMs) outperform simpler deterministic algorithms for anomaly detection when the amount of training data is limited. When learning the behavior of a parallel implementation of the *Fast Fourier Transform* (*FFT*), for example, experiments were conducted to determine the number of training samples (sequences of library function calls generated by the application) required to completely describe the program. Figure 3 contrasts the number of training samples required by the ANN and the HMM with the number of samples required by a deterministic algorithm to model this scientific application. Details of this experiment can be found in [3,4]. Clearly, the deterministic algorithm achieves a high accuracy (*i.e.* low number of false positives) with a large number of samples, but the ANN and the HMM perform better with less training data. In a system where it is difficult or expensive to gather large sets of training data and where the total number of samples from the event source that needs to be provided is difficult to determine, traditional deterministic algorithms are not appropriate.

The intelligent agents integrated into the Ganglia monitoring system are implemented as artificial neural networks and hidden Markov models. A brief discussion of the use of these machine learning techniques for solving the anomaly detection problem is presented in the next section.

5.1 Artificial Neural Network

The use of artificial neural networks (ANNs) for constructing classifiers has become popular in recent years. Compared with other approaches for classifier construction (*e.g.* template matching, statistical analysis, and rule-based decision trees) neural network models have the advantages of having relatively low dependence on domain specific knowledge and the availability of efficient learning algorithms.

5.1.0.0.1 Description An artificial neural network is composed of simple processing units and weighted connections between the units that are used to determine how much one unit will affect another. In this work, a feed-forward neural network is used to build the model of normal program behavior. The feed-forward network consists of three types of units: input units, hidden units, and output units. By assigning a value to each input unit and allowing the activations to propagate through the network, a neural network performs a functional mapping from one set of values (assigned to the input units) to another set of values (appearing in the output units). The mapping function is stored in the weights of the network. An example of a two-layer feed-forward neural network with six input units, four hidden units and a single output unit can be seen in Figure 4.

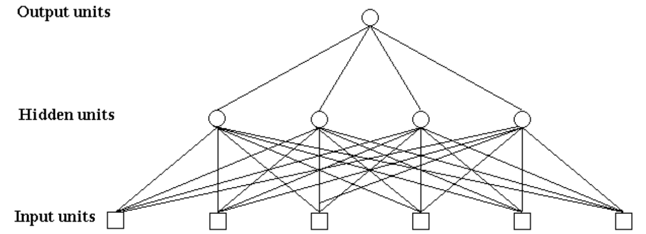

Figure 4: A two-layer feed-forward network

5.1.0.0.2 The Backpropagation Algorithm The training of artificial neural networks for anomaly detection involves the use of both normal and anomalous data. The experiments described in this paper use normal data collected from audit data and anomalous data generated by randomly injecting artificial anomalies into normal data. Anomalies were spread throughout the training space to the extent possible. The network weights were initialized to random values prior to training. A traditional backpropagation algorithm was used to find an optimal set of weights to describe the set of subsequences of calls issued by an application.

The backpropagation algorithm can be seen as a gradient descent method in weight space where the goal is to minimize the error propagated throughout the network by the input nodes and hidden nodes (See Figure 5). The error of the network is defined as the difference between the expected output (*i.e.* normal or anomalous subsequence of calls) and the output node. The key of the backpropagation algorithm is to "assess the blame for an error and divide it among the contributing weights" [25, p.579].

In order to build the lightweight detection system required for high performance cluster monitoring, computational overhead must be reduced as much as possible without compromising the accuracy of the detector (a fundamental design objective). Moreover, in Linux kernel programming, the standard C math library cannot be accessed. Therefore, a simple activation function that retains accuracy and reduces computation is needed. In order to reduce computational requirements, the standard sigmoid function (1) can be modified as suggested by Tveter [26]. This simple sigmoid function is given in (2) and its derivative is presented in (3).

$$y_{std} = \frac{1}{1 + e^{-x}} \quad (1)$$

$$y = \frac{\frac{x}{2}}{1 + |x|} + 0.5 \quad (2)$$

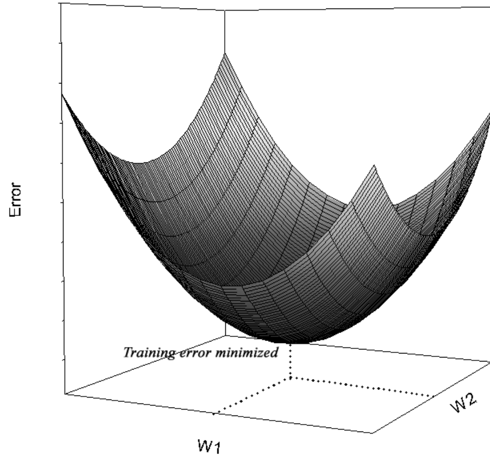

Figure 5: The backpropagation algorithm as a gradient descent search in weight space. The training error is minimized with respect to the set of weights W1 and W2

$$y' = \frac{1}{2(1+|x|)^2} \tag{3}$$

5.1.0.0.3 The Online Detection Algorithm with ANNs The main goal of the ANN-based anomaly detector described in this paper is to determine whether an input sub-sequence of calls is anomalous or normal. Therefore, a sliding window is used to divide a trace into a set of small sub-sequences. As an example, assume the following trace:

execve, brk, open, fstat, mmap, close, open, mmap, munmap

Processing this trace with a window size of 4 yields the follow subsequences:

position 3	*position 2*	*position 1*	*current*
			execve
		execve	brk
	execve	brk	open
execve	brk	open	fstat
brk	open	fstat	mmap
open	fstat	mmap	close
fstat	mmap	close	open
mmap	close	open	mmap
close	open	mmap	munmap

Each system call is represented by a binary number and the binary representations of the calls in a window are concatenated to yield an input vector. A single output node is used to indicate the status of the input sub-sequence. A value of 0 indicates a normal subsequence and a value of 1 indicates an anomalous subsequence. Because researchers have reported that anomalous sequences in operating system calls tend to occur in clusters [27,28], a scheme called *maxburstcounter* [29] was implemented. In this scheme, a counter of anomalies is incremented every time an anomalous call is found. Conversely, the counter is decremented at a slow rate when the application is behaving as expected. The two fundamental assumptions of this method are as follows:

1. An anomalous sequence seen long ago should have only a small effect on classification of the entire trace.
2. Although anomalous sequences tend to occur locally, they are not necessarily consecutive.

The main advantage of using *maxburstcounter* is that it allows for anomalous behavior that occasionally occurs during normal system operation, while simultaneously, it is sufficiently sensitive to detect the temporally co-located anomalies that occur when a program is being misused. This mechanism is similar to the *leaky bucket method* of Ghosh et al. [27]and the *LFC* method by Somayaji [11]. However, it is much less computationally expensive than these methods and enables the monitoring of a large number of operating system calls.

5.2 Discrete Hidden Markov Model

Discrete hidden Markov models are useful for anomaly detection because "...when the observations are categorical in nature, and when the observations are quantitative but fairly small, it is necessary to use models which respect the discrete nature of the data" [30, p.3]. This section introduces a brief description of stationary discrete first order hidden Markov models. For a more complete description of HMMs refer to the work of Rabiner [24] and MacDonald and Zucchini [30] among others.

5.2.0.0.4 Description Consider a system with N states $S_i, S_j, S_k ...$, where each state represents some physical (observable) event. Employing Rabiner's notation, let the actual state at time t be q_t. By assuming a discrete first order Markov chain, the probability that $q_t = S_i$, for some state S_i, is given by

$$P[q_t = S_i | q_{t-1} = S_j, q_{t-2} = S_k, ...] = P[q_t = S_i | q_{t-1} = S_j] \tag{4}$$

Furthermore, assuming a stationary process, it is not necessary to keep track of the time t to compute transition probabilities, and therefore

$$P[q_t = S_i | q_{t-1} = S_j] = a_{ij}, \quad 1 \leq i, j \leq N \tag{5}$$

This system is considered to be an "observable" first-order Markov model because each state is associated with one (and only one) observable event and the history of previous transitions is summarized in the last transition. Such a Markov chain can be seen as a finite state automata (FSA) with probabilistic transitions. An nth-order Markov chain can always be converted into an equivalent first-order Markov chain given a large state space [31]. In contrast, in a hidden Markov model, a state cannot be observed directly, since each state outputs different symbols with some probability distribution B. Therefore, a hidden Markov model is a doubly stochastic process, where the states represent an unobservable condition of the system [32]. Because this work deals with "discrete" HMMs, the output probability distribution is discrete. The elements of a discrete HMM $\lambda = (A, B, \pi)$, are described as follows:

1. N, the number of states,
2. M, the number of distinct observation symbols per state (the alphabet size),
3. A, the state transition probability distribution,
4. B, the observation symbol probability distribution, and
5. π, the initial state distribution.

5.2.0.0.5 The Baum-Welch Algorithm The Baum-Welch algorithm is an expectation maximization (EM) technique that is generally used to train the transition and symbol probabilities of an HMM utilizing the concept of forward and backward probabilities [24]. The forward procedure finds the probability of the partial observation sequence from the first

event to some event *O(t)* at time t, whereas the backward procedure finds the probability of the partial observation from *O(t+1)* to the last event in the sequence.

The forward variable corresponds to the probability of the partial observation sequence O up to time t and state S_i at time t, given the model λ. It is defined as

$$\alpha_t(i) = P(O_1 O_2 ... O_t, q_t = S_i|\lambda) \quad (6)$$

and can be computed by induction, knowing that

$$\alpha_1(i) = \pi_i b_i(O_1), \quad 1 \leq i \leq N \quad (7)$$

and

$$\alpha_{t+1} = \left[\sum_{i=1}^{N} \alpha_t(i) a_{ij}\right] b_j(O_{t+1}). \quad (8)$$

This variable can be used as an efficient estimate (in $O(N^2T)$ time) of the likelihood that the observation O is generated by the model λ:

$$P(O|\lambda) = \sum_{i=1}^{N} \alpha_T(i). \quad (9)$$

Note that $P(O|\lambda)$ can also be computed in $O(TN^T)$ time by summing the probability of occurrence of the observations $O_1 O_2 ... O_\tau$ for each hidden state sequence $q_1 q_2 ... q_\tau$ from the set Q of all possible hidden state sequences:

$$P(O|\lambda) = \sum_{Q} P(O|Q, \lambda) P(Q|\lambda) \quad (10)$$

which yields

$$P(O|\lambda) = \sum_{q_1, q_2, \dots q_\tau} \pi_{q_1} b_{q_1}(O_1) a_{q_1 q_2} b_{q_2}(O_2) ... a_{q_{\tau-1} q_\tau} b_{q_\tau}(O_\tau). \quad (11)$$

On the other hand, the backward variable is the probability of the partial observation from *t+1* to the last event, given the state S_i at time t and the model λ. It is defined as

$$\beta_t(i) = P(O_{t+1} O_{t+2} ... O_T | q_t = S_i, \lambda) \quad (12)$$

and can also be computed by induction, using the fact that

$$\beta_T(i) = 1, \quad 1 \leq i \leq N \quad (13)$$

and

$$\beta_t(i) = \sum_{j=1}^{N} a_{ij} b_j(O_{t+1}) \beta_{t+1}(j). \quad (14)$$

The Baum-Welch algorithm updates the model λ (M-step) using the probability of being in state S_i at time t and state S_j at time *t+1*, given the model and the observations. This variable, $\xi_t(i,j)$ can be seen as an efficient auxiliary variable for the E-step of the algorithm and is given by:

$$\xi_t(i,j) = \frac{\alpha_t(i) a_{ij} b_j(O_{t+1}) \beta_{t+1}(j)}{\sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_t(i) a_{ij} b_j(O_{t+1}) \beta_{t+1}(j)}. \quad (15)$$

Given $\xi_t(i,j)$, the probability of being in state S_i at time t given the model and observation sequence can be estimated as:

$$\gamma_t(i) = \sum_{j=1} \xi(i,j). \quad (16)$$

The initial state distribution can be computed as the expected frequency of being in state S_i at time $t = 1$ as:

$$\overline{\pi}_i = \gamma_1(i). \quad (17)$$

The state transition probability distribution is given by the expected number of transitions from the state S_i to state S_j divided by the expected number of transitions from state S_i:

$$\overline{a}_{ij} = \frac{\sum_{t=1} \xi_t(i,j)}{\sum_{t=1} \gamma_t(i)}. \quad (18)$$

The observation symbol probability distribution can be computed as the expected number of times state j occurred and the symbol v_k was observed, divided by the expected number of occurrences of the state j:

$$\overline{b}_i(k) = \frac{\sum_{s.t.\ O_t = v_k}^{t=1} \gamma_t(j)}{\sum_{t=1} \gamma_t(j)}. \quad (19)$$

A new estimator $\overline{\xi_t}(i,j)$ can be computed using $\overline{\pi}_i$, $\overline{a}_{ij}$ and $\overline{b}_i(k)$. This process is repeated several times until some limiting point is reached. It has been proven by Baum et al. that this process leads to an increase in the likelihood $P(O|\lambda)$ [24]. It is important to observe, however, that the maximum-likelihood procedure only results in a local maxima. The time and space complexity of the Baum-Welch algorithm is $O(N^2T)$.

5.2.0.0.6 The Threshold-Based Online Detection with HMMs Given a sequence of events O and a model $\lambda = (A, B, \pi)$, the following algorithm for detection of single anomalies was implemented:

1. Initialize the counter of anomalies, $C = 0$.
2. For each event O_t ($0 \leq t < T_O$)
   (a) For each state S_j that can be reached from the previous state, *i.e.* if the probability a_{ij} of moving to the current state from some state S_i is greater than the threshold θ_A
   - If the probability of producing the symbol O_t in the current state, $b_j(O_t)$, is less than a user threshold θ_B, then the event is considered anomalous, and $C = C+1$.
   - Otherwise, repeat step 2.

   (b) If O_t could not be produced by any state then $C = C+1$.
3. Output C.

6 Integration of Intelligent Anomaly Detection Agents into Ganglia

An event-driven system architecture has been chosen where specific sensors collect information every time a discrete event is generated. Sequences of those events are used for training the intelligent agents. In this research, the set of discrete events considered corresponds to the application library function calls and/or operating system calls issued by a UNIX/Linux parallel application.

6.1 Operating System Calls (Kernel-level Monitoring)

System call interfaces are the application programmer interfaces (APIs) through which the operating system kernel provides low-level operations (such as memory allocation, file access, and network transfers) to the user space applications. For Linux, this interface is wrapped by the GNU standard C library (*libc*) so that C programmers can have a common standard interface. Although user space applications usually utilize low-level operations through library function calls, some

applications and attacks can bypass the *libc* interface and invoke system calls directly [2].

Several mechanisms can be used to obtain system call information. The current implementation uses a loadable kernel module (LKM) because of the flexibility, efficiency and compatibility this approach provides. LKMs are used by the Linux kernel and many other UNIX-like systems as well as Microsoft Windows to expand their functionality. An LKM can access all the variables in the kernel. This method is more flexible than the methods based on *kernel patching* because it does not require changes to the kernel source code. A privileged user can load or unload the LKM during runtime. The disadvantage is that an attacker can install malicious code into the kernel before the tool is loaded. This weakness can be resolved by adding the tool into a startup script.

Extensions that use kernel level interposition have a broad range of capabilities. For example, extensions "...can provide security guarantees (for example, patching security flaws or providing access control lists), modify data (transparently compressing or encrypting files), re-route events (sending events across the network for distributed systems extensions), or inspect events and data (tracing, logging)" [33, p.1]. The primary advantage of this method is that it cannot be bypassed. All programs must use the system call interface to access the low-level functionality implemented by the kernel. Another advantage is it does not require modification of the application code. Only the kernel needs to be changed. The disadvantage is that a security hole in the monitor program is much more dangerous than with other methods. It can cause the system to crash or to grant root privilege.

It is interesting to note here that fundamental design objectives in early attempts to develop secure operating systems required the security mechanisms to be tightly coupled at the kernel level, and to be non-bypassable, tamper proof, and verifiably correct [34]. Those same characteristics should be included in any secure low-level monitoring system.

6.2 Library Function Calls (User-level Monitoring)

A software application issues calls to the operating system to perform a wide variety of functions such as I/O access, memory requests, and network management. However, many application programmer interfaces (APIs) do not make use of system calls, mainly for performance reasons or because no privileged resources need to be manipulated. A typical example is the set of functions such as *cos*, *sin* or *tan* from the standard mathematical library of C.

Linux provides a large collection of mechanisms to trap system and function calls from any process in user-mode. For instance, in order to monitor kernel calls, the OS provides the tools *strace*, *trace* and *truss* [35], but these tools only record kernel level functions, and the trap mechanism produces too much overhead [36]. However, another method that has been widely used for implementing performance and monitoring tools is library interposition.

The link editor (*ld*) in a Linux operating system builds dynamically linked executables by default. Building "incomplete" executables, the link editor allows the incorporation of different objects in real time. Communication between the main program and the objects is done by shared memory operations. Such (shared) objects are called dynamic libraries: "A dynamic library consists of a set of variables and functions which are compiled and linked together with the assumption that they will be shared by multiple processes simultaneously and not redundantly copied into each application" [36, p.1].

In the Linux system, the link editor uses the LD_PRELOAD environment variable to search for the user's dynamic libraries. Using this feature, the operating system gives the user the option of interposing a new library. Interposition is "the process of placing a new or different library function between the application and its reference to a library function" [36]. Thus, the library interposition technique allows interception of the function calls without the modification or recompilation of the dynamically linked target program. By default, C compilers in Linux use dynamic linking. Furthermore, "most parts of the Linux libc package are under the Library GNU Public License, though some are under a special exception copyright like *crt0.o*. For commercial binary distributions this means a restriction that forbids statically linked executables" [37].

The user functions (*i.e.,* the functions inside the interposition library with the same prototype as the "real" ones) are able to check, record and even modify the arguments and the response of the original function call. Other advantages include the capability to profile a subset of the library rather than the entire library, the ability to generate levels of profiling, and provision of control over nesting levels inside the library [36].

The main disadvantage of the interposition library technique is that it can be bypassed by calling functions at a lower level (for instance, executing system call interruptions) [38]. Also, as explained above, the process of searching the symbol table in the new interposition library and the allocation of memory increases the execution time of the target process.

The libraries to be profiled are defined by the system administrator using a configuration file and template like the one depicted in Figure 6. The source code needed to intercept function calls from any dynamic library is generated automatically.

6.3 System Architecture

Figure 7 contrasts the mechanisms used to collect and analyze operating system calls and library function calls issued by a dynamically linked program. The output of the detectors is sent to the Ganglia monitoring system via the interface *gmetric*. Four important characteristics of the system should be noted. First, operating system calls describe low-level access of the application and cannot be bypassed by a malicious user. In contrast, application library function calls represent high-level information about the application and can be bypassed. Second, monitoring of function calls is more efficient than monitoring of system calls, because the number of operating system calls greatly outnumbers the number of function calls. Third, *root* access is needed to monitor system calls because the system call wrapper must be executed as a part of the kernel. In contrast, function calls can be monitored in user space, and *root* access is not required. Finally, although our current experiments only collect calls from the libraries *libc* and *libmpipro*, other dynamic libraries can be easily included in the detection scheme.

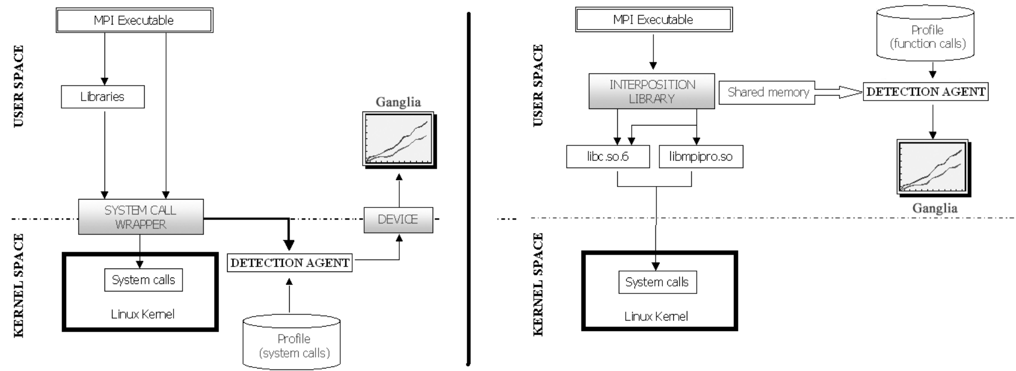

Figure 7: Architectures used to monitor system calls (in kernel space) and function calls (in user space)

```
__FUNCTYPE __MPIAPI_C __FUNCREALNAME (__FUNCPARAM)
{
    typedef __FUNCTYPE(*function_type) (__FUNCPARAM);
    static function_type function=NULL;
    static char* function_name=__FUNCNAMESTRING;

    __TYPERETVAL __DEFINERETVAL

    if (!function) {
        __HANDLEMPILIBRARY
        __OPENMPILIBRARY
        function = (function_type) dlsym(__HANDLER,function_name);
        __CLOSEMPILIBRARY
    }

    if ((!ThisLibraryCall) && (DoProfile)) {
        ThisLibraryCall=TRUE;
        //execute the funtion and then profile
        __ASSIGNRETVAL ((*function)(__FUNCNOTYPEPARAM));
        PROFILE(__FUNCID,
            __FUNCTOLOGPARAM);
        ThisLibraryCall=FALSE;
    }
    else //do not profile, only execute
        __ASSIGNRETVAL ((*function)(__FUNCNOTYPEPARAM));

    __RETURNRETVAL
}
```

Figure 6: Template to intercept any C function

7 Experiments

7.1 Dataset

Five MPI (Message Passing Interface) parallel applications written in C were selected for the test cases. These were executed on a small cluster with Linux RedHat 7.1 (kernel 2.4.2). The cluster contains one head node able to compile and launch the parallel programs and eight computing nodes. The head node is a four-CPU SMP computer and the other nodes are dual-CPU SMP computers. The machines are fully connected with Ethernet and Giganet switches and the MPI environment used in all the experiments was MPI/Pro 1.5. Only the head node can be accessed from the Internet.

The following applications were selected to test the integration of the intelligent anomaly detectors with Ganglia: an implementation of the Fast Fourier Transform (FFT) [39], an NPB benchmark based on a bucket sort (IS) [40], an implementation of the LU Factorization method for solving systems of linear equations (LU), a modified version of LLCbench from the MPBench benchmarks suite (LL) [41], and the Network Protocol Independent Performance Evaluator (NETIPPE) [42]. Training and test data for the anomaly detectors were generated by executing each program fifteen times with different input parameters and collecting both the sequence of function calls and system calls issued by each executable in each of the four nodes of the cluster. As an example, when executing FFT, a total of 60 function-call trace files and 60 system-call trace files were created.

Previous research by Tan and Maxion [43] among others have demonstrated that the intrinsic structure of the data, as measured by conditional entropy, affects the performance of anomaly detection algorithms. Entropy is associated with the level of uncertainty or randomness in the sequence of observations. Because entropy has no upper bound, a relative entropy is often used where a minimum value of 0 means that the sequence is completely predictable and a maximum value of 1 indicates that the sequence is completely random. Of particular interest is the sequential dependence of the data where some events precede others. This property can be measured as the conditional relative entropy (CRE) of the source. Suppose X and Y are two random variables. Knowing the conditional probability distribution $P(Y|X)$, the CRE can be computed as

$$CRE = \frac{-\sum_x P(x) \sum_y P(y|x)\ log\ P(y|x)}{MaximumEntropy} \tag{20}$$

where $MaximumEntropy$ is the entropy of a theoretical source in which all symbols have the same probability. $P(Y|X)$ can be defined as a matrix M, in which the rows represent the current symbol and the columns represent the next symbol in the sequence: The value of $M(x,y)$ is the conditional probability $P(Y=y|X=x)$.

Figure 8 shows the conditional relative entropy for each program. An important conclusion is that the CRE of the function call sequences varies from 0.05 to almost 0.35, indicating that some of the programs are highly deterministic (such as NETPIPE), but others exhibit more complex behavior. In contrast, the CRE of the system call datasets for all the applications shows far less variation for all values (in the range 0.25 - 0.3).

Since the system call traces record more detailed events for each program operation (such as I/O transactions or memory allocation), the number of calls invoked by a program is quite large and, as indicated by the CRE values, the sequences of calls show less organization than is often seen with function calls. This can make it more difficult to accurately model the behavior of the program. For example, consider the system calls generated by high-level function calls such as MPI_Send and MPI_Recv. Both calls will make use of similar low-level system requests for operations such as memory allocation, input/output device control and timestamps. Figure 9 presents histograms of the frequency of different calls (both system calls and function calls) observed for 20 samples of FFT and LU. It is difficult to differentiate the behavior of the two programs based on system call logs, but in the function call logs, there are library function calls that are executed exclusively by one of the programs and not the other.

It is also important to observe the difference in the total number of library function calls and operating system calls issued by the applications. A summary of the average number of calls issued by each application is shown in Figure 10. Clearly, the number of system calls is substantially greater than the total number of function calls for most of the applications. This may affect both the performance and accuracy of the intelligent detection systems. For a comparison of accuracy and overhead between user-level and kernel-level monitoring refer to the author's previous work [3, 4].

7.2 Configuring the Intelligent Detection Agents

In all of the experiments reported in this paper, ANN-based detectors were used for kernel-level monitoring, and HMM-based detectors were used for user-level monitoring. Each type of model has a different set of parameters that must be determined.

An appropriate number of states for the hidden Markov model must be estimated from the training dataset in order to generate high-quality *profiles* for the parallel applications. If an HMM has fewer states than needed, or if the topology is not adequate, the estimation algorithm will generate a suboptimal model. For example, Figure 11 shows the average Baum-Welch likelihood estimation of five HMMs from 60 samples of sequences of library calls for *FFT* using different numbers of states. The *y-axis* shows the log of the probability of the sequence being generated by each of the models (the highest point on this axis corresponds to a probability of 1, *i.e.* exact match between the 60 logs of function calls and

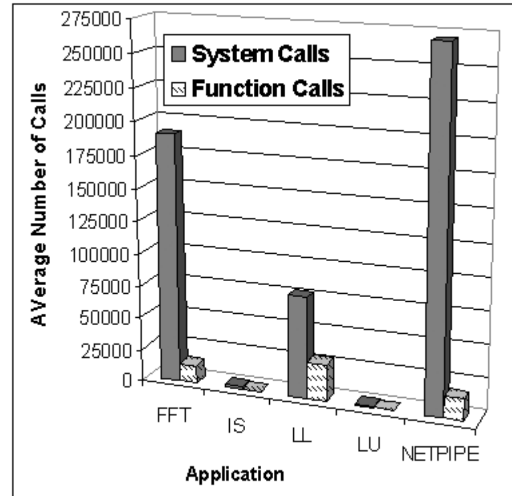

Figure 10: Average number of calls for each MPI application.

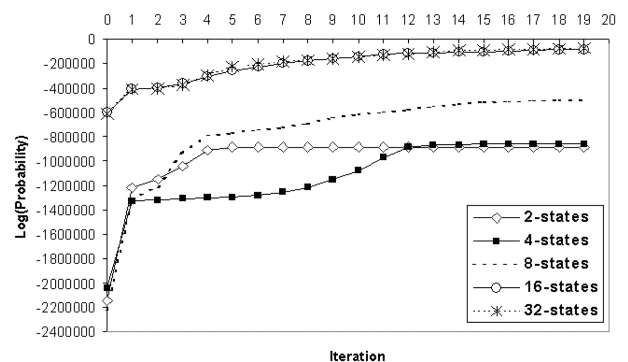

Figure 11: Baum-Welch estimation of 60-function call logs for FFT

the HMMs). This graph shows that there is a large improvement in the likelihood by using 16 states but little additional improvement using 32 states. In the experiments reported in this paper, HMMs with 16 states were used and up to 20 iterations of the Baum-Welch algorithm were allowed in training. Also, the empirical thresholds $\theta_A = \theta_B = 0$ have been selected for the online detection algorithm with the HMM.

The number of hidden nodes and number of epochs influence the classification performance of neural networks. A five-fold cross-validation technique was used to select the best number of hidden nodes and the most appropriate number of epochs to prevent overfitting. Results of this experiment resulted in the selection of 32 hidden nodes. The number of epochs was controlled to prevent memorization (and therefore lack of generalization) of the patterns. The subsequence length (window size) of the input examples for the feed-forward network was set to six.

7.3 Monitoring of Applications

Figure 12 shows the percentage of CPU time spent by user processes, the average load for the last one minute, and the amount of free memory of four nodes in the cluster, when five FFT applications with different parameters were executed consecutively. This data was collected by pulling data

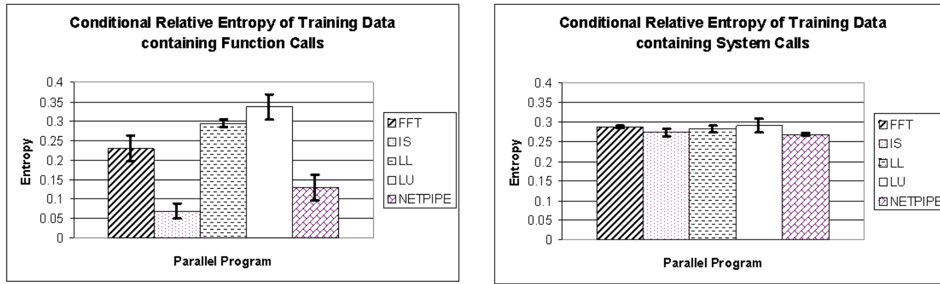

Figure 8: Conditional Relative Entropy of Training Data

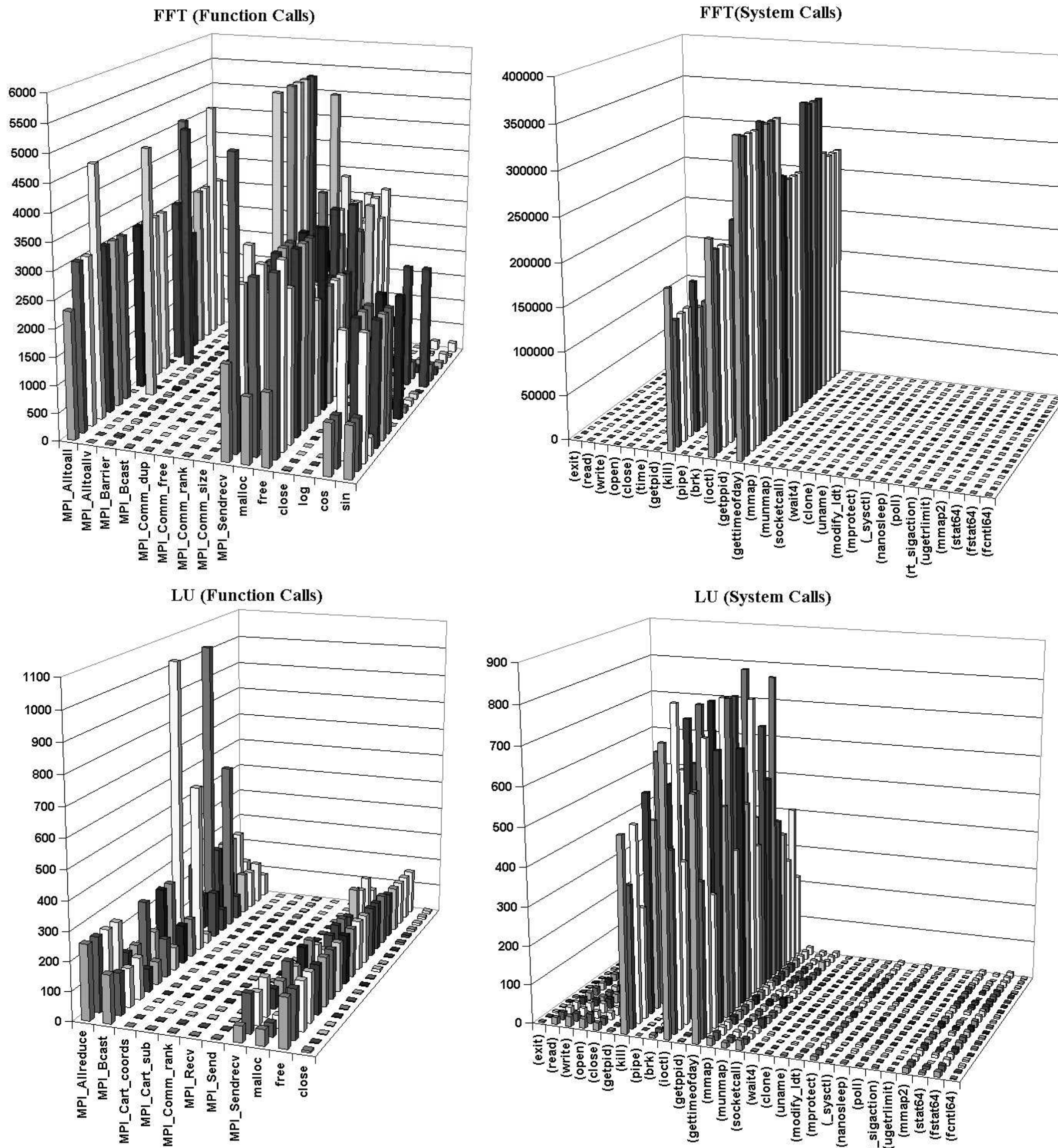

Figure 9: Frequency of calls issued by 20 samples of FFT and LU

from *gmond* every five seconds by executing the command *telnet localhost 8649* and parsing the corresponding XML file. These results demonstrate that although Ganglia monitoring is useful for determining the overall status of the distributed system, it can be misleading when it is used to monitor a particular parallel application. For example, the execution of a parallel program with different input data and parameters will result in very skewed values for CPU usage times, and therefore, it is difficult to summarize this feature using simple statistical properties such as mean and variance. The problem of profiling applications with traditional Ganglia metrics is even more difficult when the application makes use of randomized algorithms. Because most of the sensors used in Ganglia are time-driven (*i.e* a snapshot of the sensor is taken at a given time period), creating a useful model of the sensor for a complex distributed application can be a difficult task.

An experiment was conducted to demonstrate the added detection capability provided by our intelligent sensors. Consider a hypothetical distributed environment where a single application is allowed to execute. Any other program is considered to be anomalous even if it has been launched by an authorized user. Figure 13 shows three standard Ganglia metrics along with the output of one intelligent anomaly detection agent (an HMM analyzing library function calls) for a hypothetical environment where the application FFT is the only one allowed to run. In this particular example, four FFT programs were launched with different parameters along with one IS program used to simulate the execution of an unauthorized program. A system administrator can easily recognize when a user is misusing the system by looking at the output of the intelligent anomaly detector. Note that in this example one can observe a small reduction in the amount of available memory that indicates the existence of potential anomalies in the system. However, this reduction in free memory could also be attributed to other factors such as incorrect arguments for the program or a small input dataset.

7.4 Integrating Anomaly Detection with the Ganglia Front End

Previous experimental results demonstrated that the output of an anomaly detection system can help a system administrator determine when an application is not behaving as expected. The next step is the integration of the detectors into the Ganglia framework. This is easily accomplished by using the *gmetric* interface and by modifying the PHP source of the Ganglia front end. Three test cases were conducted to demonstrate the capabilities of the system with intelligent anomaly detection agents integrated into the Ganglia monitoring system. The first test case shows how the execution of an unauthorized program can be detected using the integrated system. The second test case shows how the integrated system can be used to recognize network or software faults in the system. The third test case demonstrates simultaneous use of user-level and kernel-level monitoring for detecting anomalies.

Case Study 1. Detection of Unauthorized Applications: A test environment was created where it was assumed that only one application (IS) is authorized to run and the execution of other programs such as LL, FFT, LU and NETPIPE should be detected as anomalies. Figure 14 shows the output of Ganglia from the most recent 10 minutes of activity in the cluster when both legal and illegal programs have been executed. Note that the figure shows the output of the user-level monitoring as *AD_FC* (*i.e.*, anomaly detection with function calls in the Ganglia framework) for six computing nodes, as well as four standard Ganglia metrics such as CPU Load and bandwidth usage. As was demonstrated previously, it is difficult to determine whether unauthorized applications are being executed in the cluster using only the traditional Ganglia metrics. On the other hand, the six anomaly detectors shown in Figure 14 (each one monitoring a single node in the cluster) alert the system administrator of suspicious activity in nodes 1, 2, 6 and 7. Furthermore, the figure indicates that some unauthorized applications were executed in four nodes, while some other unauthorized applications employed only nodes 1 and 2.

Similar techniques can be applied on a distributed system in order to detect exploitation of computer resources by malicious users who are attempting to gain access to sensitive information or to perform distributed network attacks. The attack on academic supercomputing laboratories using simple MPI applications reported in 2004 [1] is an example of the type of attack that requires more sophisticated sensors than those provided by firewalls and traditional intrusion detection systems.

Case Study 2. Fault Tolerance and Detection: Software and hardware faults are likely to occur during long-running jobs in a cluster with several individual nodes. Several approaches for the development of fault-tolerant distributed applications are known (see for example the previous work on fault tolerance in MPI programs [44,45]). A simple mechanism that checks the return status of MPI functions was implemented. If the return code is not equal to MPI_SUCCESS, an error is reported to a log via the standard C function *fwrite*.

In this particular scenario, an interposition library was created to simulate anomalies in the network interface when the application LLCbench was executed five times, with a 0.001 probability of error for the functions MPI_Send and MPI_Recv. This is an example of fault injection and simulation of attacks described in previous work [3,4]. Every time a fault is reported via the *fwrite* call by a parallel program, the intelligent agents detect the change in the sequence of observations and inform Ganglia that the counter of anomalies has been increased. Figure 15 shows the output of the anomaly detectors in the Ganglia front end. The intelligent detection agents integrated in the Ganglia front end clearly provide a simple way to visualize system faults. Again, these faults are difficult to detect using the standard Ganglia metrics.

Note that although fault detection is not widely used in MPI programs, similar *catch-throw* mechanisms often occur in high-quality software. Therefore, recognizing a change in the sequence of discrete events can still be used as a means of detecting faults.

Case Study 3. Simultaneous Detection of Anomalies in User Space and Kernel Space: The author's previous work conducted experiments to compare the accuracy in the detection and the overhead produced by the intelligent detection agents [3, 4] . However, the behavior of distributed applications when both systems are simultaneously used to detect anomalies in real-time has not been observed before. Figure 16 shows the detection of network faults in a single node of the cluster when both user-level monitoring and kernel-level monitoring are enabled.

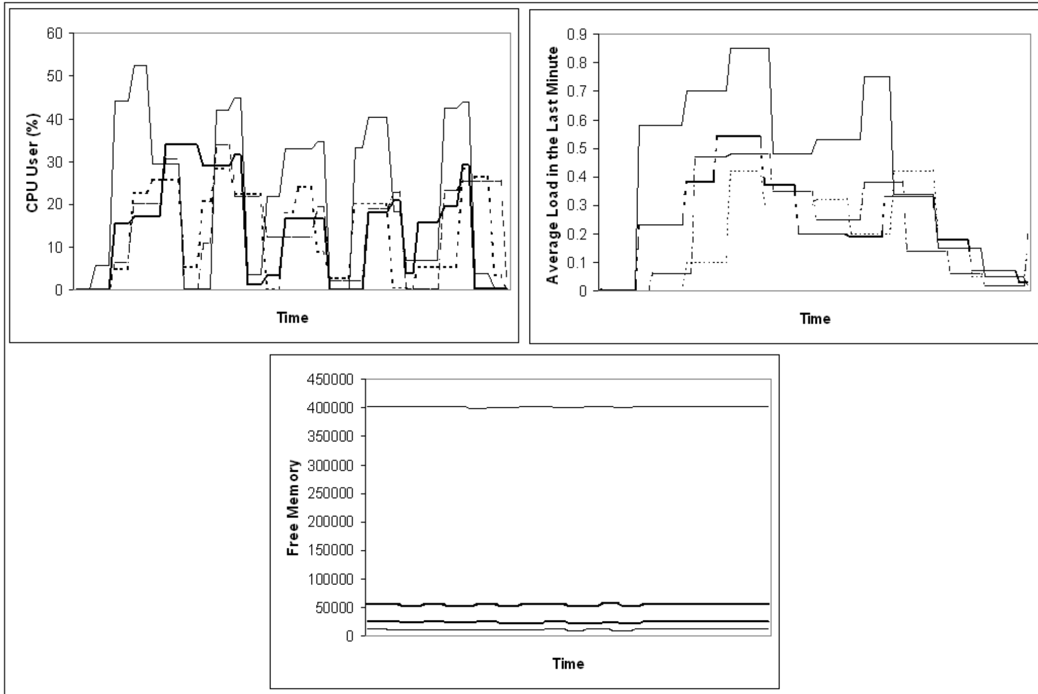

Figure 12: Three standard Ganglia measures for five consecutive executions of FFT with different parameters. It is difficult to visually recognize the most important characteristics of the application

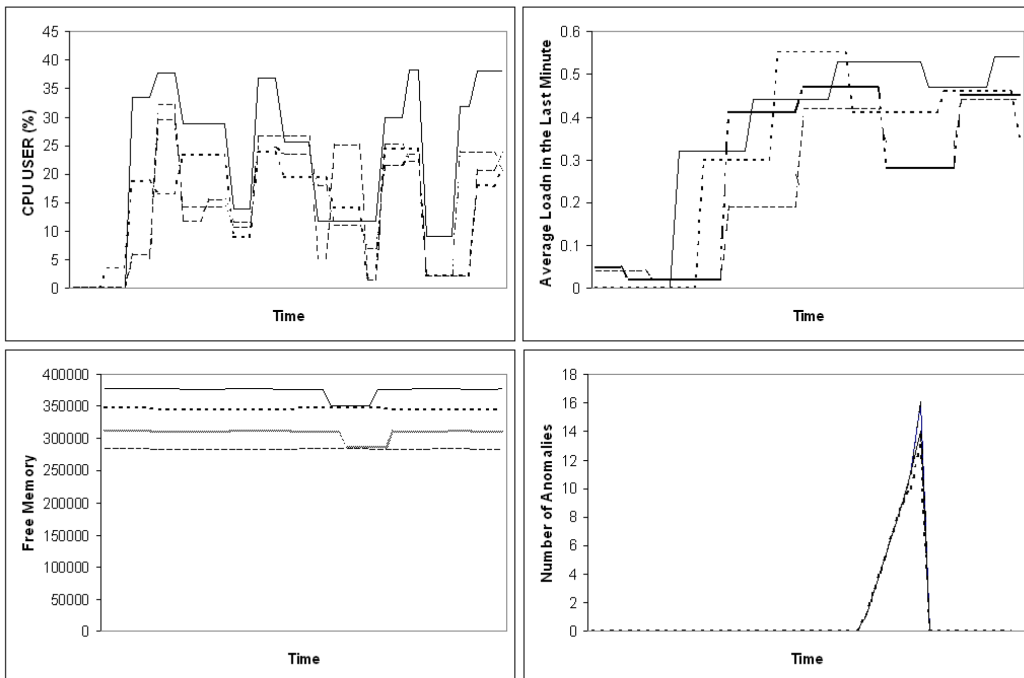

Figure 13: Output of three Ganglia sensors and one intelligent anomaly detection agent for five consecutive program executions (four of FFT one of IS) in four nodes. In this hypothetical environment, FFT is the only legal application and the execution of IS (fourth program executed) should be detected as an anomaly

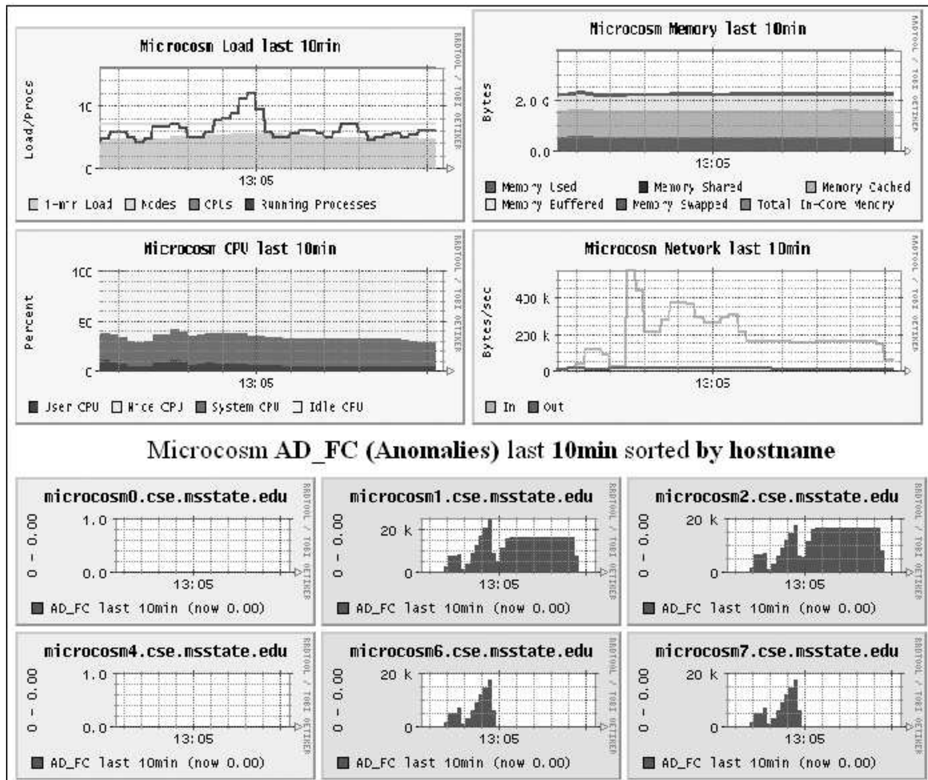

Figure 14: Detection of unauthorized applications using the Ganglia front end enhanced with an intelligent anomaly detector (AD_FC)

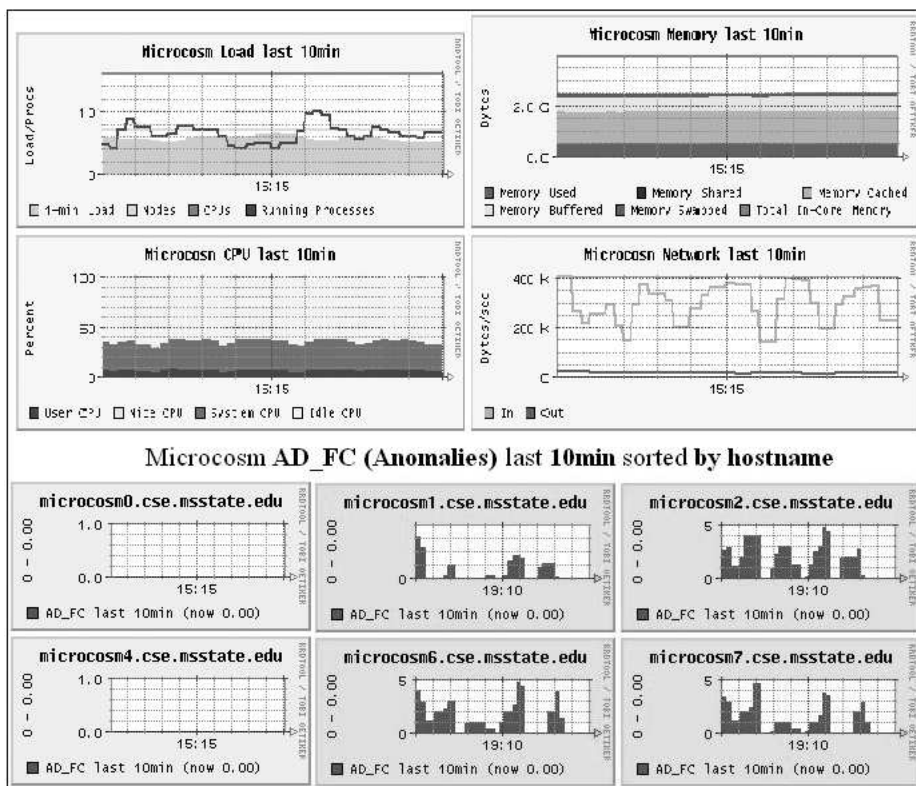

Figure 15: Detection of network faults using the Ganglia front end enhanced with an intelligent anomaly detector (AD_FC)

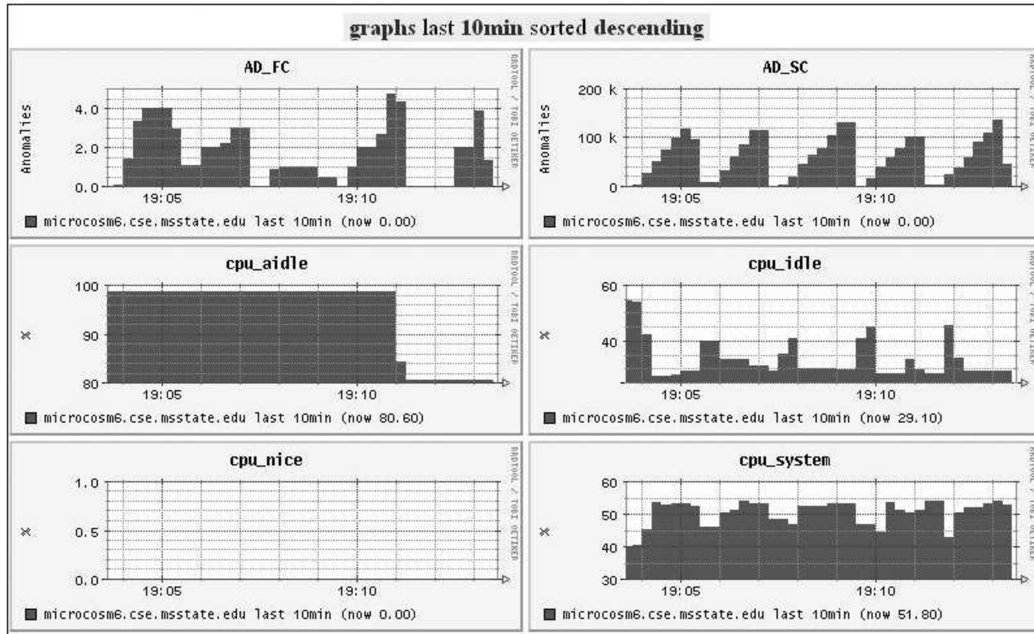

Figure 16: Simultaneous detection of network faults in a single cluster node for the previous 10 minutes using the HMM-based detector and the ANN-based detector. The outputs in the first row correspond to user-level monitoring (*AD_FC*) and the kernel-level monitoring (*AD_SC*). Four built-in Ganglia metrics are also shown.

One of the problems encountered when executing both systems at the same time is contamination of the system call sequences by calls made by the user level monitoring system. Every operating system call issued by the user space monitoring system is recorded by the kernel space monitoring system. Therefore, even if an MPI application is behaving as expected (and the user-level monitoring system reports 0 anomalies), the kernel-level detection system will detect changes in the sequence of operating system calls and report these as anomalies. To solve this problem, a new system call exclusively executed by the user-level monitoring system that acts as a *flag* to temporarily deactivate the detection of system calls was created. Although this naive solution allows observation of the behavior of both systems in a research environment, it is not an adequate solution for real-world systems because it can create a security vulnerability. Further research is needed to develop better mechanisms for allowing the simultaneous execution of the two levels of detection in real-time including studies of the security, scalability and performance of such mechanisms.

7.5 Limitations of Anomaly Detection with Sequences of Discrete Events

The fundamental assumption of this work is that high-quality profiles (artificial neural networks or hidden Markov models) can be used to discriminate between normal and suspicious executions of a distributed application. However, there are situations in which this assumption may not hold. For example, when the behavior of legal applications is quite varied resulting in a complex model, it may not be possible to detect the execution of illegal applications with simple behavior that is a subset of the behavior of the legal application. This phenomenon was observed in our experiments when we used one application as the "legal" application. When the model of the legal application was run against sequences of system calls from other applications that should have been detected as anomalies, it was not always possible to detect the anomalies. Figure 17 shows the results of such an experiment in which HMM profiles were created using five different programs. Each profile was run with system calls resulting from running the other applications and the percentage of anomalous function calls (*i.e., y-axis*) was computed with each profile. The programs run were (FFT, IS, LU, LL and NETPIPE). Points on the *x-axis* represent executions of different instances of each program. When the profile for one program is used, the system calls from all other programs should be flagged as anomalous. From the figure it is clear that most of the profiles are able to discriminate normal behavior (*i.e.* the intelligent agents output 0% anomalies when the parallel program being executed has the same behavior as the *profile* of such a program) from suspicious activities in the system. However, when NETPIPE is the "anomalous application", only the profiles obtained from sampling FFT and IS detect it as anomalous. NETPIPE is an example of a highly deterministic benchmark application where a large portion of the computation consists of calls to the functions MPI_Send (send a message to a specific node) and MPI_Recv (receive a message). These sequences of calls are also likely to occur in the other applications. Figure 18 repeats the same experiment with the ANN-based detector monitoring operating system calls. It is important to observe that the large difference in the scale of the *x-axis* for Figures 17 and 18 is due to the large number of operating system calls executed by the applications.

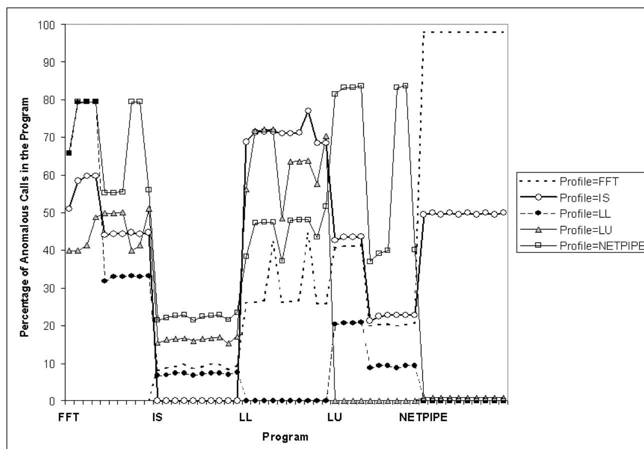

Figure 17: The percentage of anomalous function calls detected for each application execution by the HMM-based detector monitoring library function calls for a given *profile* is plotted for a work session consisting of the consecutive execution of several MPI programs.

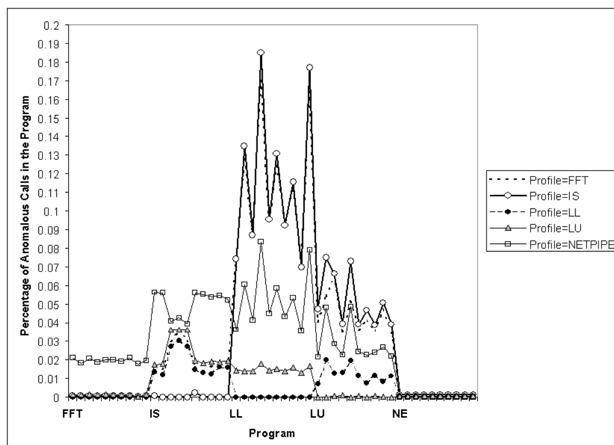

Figure 18: The percentage of anomalous function calls detected per application execution by the ANN-based detector monitoring operating system calls for a given *profile* is plotted for a work session consisting of the consecutive execution of several MPI programs.

8 Conclusions and Future Work

The central focus of the research reported in this paper is the integration of intelligent anomaly agents and high performance monitoring systems. An event-driven system architecture is used in which sensors collect information every time a discrete event is generated. The intelligent agents used for this study use machine learning techniques to develop profiles of normal behavior as seen in the sequences of operating system calls generated by an application and the sequence of function calls. The intelligent agents are able to flag anomalies that are difficult to detect in traditional monitoring systems, while the monitoring systems provide the framework needed for communication and visualization of system status. The Ganglia monitoring system was used as a test bed for integration case studies. Mechanisms provided by Ganglia make it relatively easy to integrate the new detectors into the system and to visualize the results of these detectors. The experimental results demonstrate that the integrated intelligent agents can detect the execution of unauthorized applications and faults that are not obvious in the standard output from the monitoring system. Both hidden Markov models working in user space and neural network models working in kernel space are shown to be effective. Simultaneous monitoring in both user space and kernel space is also demonstrated.

The research reported in this paper provides preliminary evidence that intelligent anomaly detectors can be effectively integrated in a high performance monitoring system. Our system can also be used to detect anomalies in general purpose systems because we use standard Linux libraries and the Ganglia framework. However the system may suffer from a large number of false positives in general environments because it becomes difficult to collect an adequate set of samples to train the models.

A number of research issues remain. More effective methods are needed for resolving conflicts encountered during simultaneous user and kernel space monitoring. Studies need to be conducted in order to determine the performance penalty incurred by the anomaly detectors and to determine scalability of the system.

Also, although previous work has shown that the anomaly detectors have small overheads [3,4], the authors are currently investigating the use of reconfigurable co-processors to reduce this overhead further. Research also needs to be conducted in the use of extended models that are able to monitor a set of attributes of the calls such as return value or call parameters. This kind of model can potentially provide more information about the expected behavior of an application.

Future work also includes the study of incremental algorithms able to estimate the parameters and topology of HMMs to reduce the space complexity from $O(N^2T)$ to $O(N^2)$. Such an algorithm will provide a mechanism for learning directly from the sequence of calls generated by the applications without the need to store and transfer large amounts of data and can be modified to handle non-stationary distributions, providing a solution for the problem of concept drift. Furthermore, in the case that it is necessary to avoid contaminating the model with outliers (elements in the dataset that do not seem consistent with past observations, including perhaps system anomalies), the incremental learning algorithm can decide whether a observation needs to be included in the model or not.

Finally, a related issue in high-performance cluster monitoring is the fusion of monitoring results by disparate sensors in a highly efficient and accurate manner. Experimental work in this area includes the use of fuzzy cognitive maps (FCMs) as the technical basis for a fusion engine. The interested reader is invited to review Siraj's work [5, 6].

Acknowledgment

This work was supported by the Naval Research, contract N00014-01-0678; the Army Research Laboratory, contract DAAD17-01-C-0011; and the National Science Foundation, contract CCR-0098024, 98-1849 and 99-88524. We also thank the members of the Center for Computer Security Research (CCSR) at Mississippi State University.

References

[1] R. Lemos, Academics search for clues after supercomputer attacks, http://news.zdnet.co.uk/internet/security/0,39020375,39152264,00.htm (April 2004).

[2] M. Torres, R. Vaughn, S. Bridges, G. Florez, Z. Liu, Attacking a high performance computer cluster, in: Proceedings of the 15th Annual Canadian Information Technology Security Symposium, Ottawa, Canada, 2003.

[3] G. Florez, Z. Liu, S. Bridges, A. Skjellum, R. Vaughn, Lightweight monitoring of MPI programs in real-time, Concurrency and Computation: Practice and Experience 17 (3) (2005) 1547–1578.

[4] G. Florez, Z. Liu, S. Bridges, A. Skjellum, R. Vaughn, Detecting anomalies in high-performance parallel programs, Journal of Digital Information Management 2 (2) (2004) 44–47.

[5] A. Siraj, S. Bridges, , R. Vaughn, Fuzzy cognitive maps for decision support in an intelligent intrusion detection system, in: International Fuzzy Systems Association/ North American Fuzzy Information Processing Society (IFSA/NAFIPS) Conference on Soft Computing, Vancouver, Canada, 2001.

[6] A. Siraj, R. B. Vaughn, S. M. Bridges, Decision making for network health assessment in an intelligent intrusion detection system architecture, International Journal of Information Technology and Decision Making 3 (2).

[7] M. L. Massie, B. N. Chun, D. E. Culler, The Ganglia distributed monitoring system: Design, implementation, and experience, Accepted for publication in Parallel Computing, http://ganglia.sourceforge.net/talks/parallelcomputing/ganglia-twocol.pdf (2004).

[8] S. Forrest, S. A. Hofmeyr, A. Somayaji, T. A. Longstaff, A sense of self for unix processes, in: Proceedings of the 1996 IEEE Symposium on Security and Privacy, Los Alamitos, CA, 1996, pp. 120–128.

[9] H. Steven A, S. Forrest, A. Somayaji, Intrusion detection using sequences of system calls, Journal Of Computer Security 6 (3) (1998) 151–180.

[10] P. A. Porras, P. G. Neumman, Emerald: Event monitoring enabling responses to anomalous live disturbances, in: Proceedings of the 20th National Information Systems Security Conference, 1997.

[11] A. Somayaji, Operating system stability and security through process homeostasis, Ph.D. thesis, The University of New Mexico, Albuquerque, New Mexico (July 2002).

[12] L. Eschenaur, Imsafe: Immune security architecture, World Wide Web, http://imsafe.sourceforge.net/inside.htm (2001).

[13] C. Warrender, S. Forrest, B. A. Pearlmutter, Detecting intrusions using system calls: Alternative data Models, in: Proceedings of the IEEE Symposium on Security and Privacy, 1999, pp. 133–145.

[14] G. Florez, Analyzing system call sequences with Adaboost, in: Proceedings of the 2002 International Conference on Artificial Intelligence and Applications (AIA), Malaga, Spain, 2002.

[15] Z. Liu, G. Florez, S. Bridges, A comparison of input representations in neural networks: A case study in intrusion detection, in: Proceedings of the 2002 International Joint Conference on Neural Networks (ICJNN'02), Honolulu, Hawaii, 2002.

[16] T. Lane, Hidden Markov Models for human/computer interface modeling, in: IJCAI-99 Workshop on Learning About Users, 1999, pp. 35–44.

[17] R. Rabenseifner, Automatic MPI counting profiling, in: Proceedings of the 42nd Cray User Group Conference, CUG SUMMIT 2000, 2000.

[18] J. S. Vetter, B. R. de Supinski, Dynamic software testing of MPI applications with Umpire, in: SC2000: High Performance Networking and Computing Conference, ACM/IEEE, 2000.

[19] M. Marzolla, A performance monitoring system for large computing clusters, in: Proceedings of the Eleventh Euromicro Conference on Distributed and Network-Based Processing, Genova, Italy, 2003, pp. 393–400.

[20] M. Lyu, V. Mendiratta, Software fault tolerance in a clustered architecture: Techniques and reliability modeling, in: Proceedings 1999 IEEE Aerospace Conference, Snowmass, Colorado, 1999, pp. 141–150, vol.5.

[21] C. M. P. Augerat, B. Stein, Scalable monitoring and configuration tools for grids and clusters, in: Proceedings of the 10th Euromicro Workshop on Parallel, Distributed and Network-based Processing, Canary Islands, Spain, 2002.

[22] R. Buyya, Parmon: A portable and scalable monitoring system for clusters, Software: Practice and Experience 30 (7) (2000) 723–739.

[23] M. Sottile, R. Minnich, Supermon: A high-speed cluster monitoring system, in: Proceedings of Cluster 2002, 2002.

[24] L. Rabiner, A tutorial on Hidden Markov Models and selected applications in speech recognition, in: Proceedings of the IEEE, Vol. 77 of 2, 1989, pp. 257–286.

[25] S. Russel, P. Norvig, Artificial Intelligence: A Modern Approach, Prentice Hall, 1995.

[26] D. R. Tveter, Backpropagator's review, http://www.dontveter.com/bpr/bpr.html (August 2002).

[27] A. K. Ghosh, A. Schwartzbard, M. Schatz, Learning program behavior profiles for intrusion detection, in: Proceedings of the 1st USENIX Workshop on Intrusion Detection and Network Monitoring, Santa Clara, California, 1999, pp. 51–62.

[28] W. Lee, S. Stolfo, Data mining approaches for intrusion detection, in: Proceedings of the 7th USENIX Security Symposium, San Antonio, Texas, 1998, pp. 79–94.

[29] Z. Liu, S. Bridges, R. Vaughn, Classification of anomalies traces of privileges and parallel programs by neural networks, in: Proceedings of the 2003 FUZZIEEE Conference, St. Louis, MO, 2003.

[30] I. MacDonald, W. Zucchini, Hidden Markov and Other Models for Discrete-valued Time Series, Monographs on Statistics and Applied Probability, Chapman and HALL/CRC, 1997.

[31] J. Bilmes, What HMMs can do, Tech. Rep. UWEETR-2002-0003, Department of Electrical Engineering, University of Washington (January 2002).

[32] G. Sun, H. Chen, Y. Lee, C. Giles, Recurrent neural networks, Hidden Markov Models and stochastic grammars, in: Proceedings of International Joint Conference on Neural Networks, San Diego,CA, 1990, pp. 729–734.

[33] T. Mitchem, R. Lu, , R. O'Brien, Using kernel hypervisors to secure applications, in: Proceedings of Annual Computer Security Application Conference, San Diego, California, 1997, pp. 175–181.

[34] Department of Defense, Trusted Computing systems Evaluation Criteria, DoD Standard 5200.28 (1985).

[35] B. Kuperman, E. Spafford, Generation of application level audit data via library interposition, Technical Report TR 99-11, COAST laboratory, Purdue University (1998).

[36] T. W. Curry, Profiling and tracing dynamic library usage via interposition, in: Proceedings of the USENIX 1994 summer conference, 1994.

[37] S. Goldt, S. van der Meer, S. Burkett, M. Welsh, The linux programmer's guide, World Wide Web, http://ibiblio.org/mdw/LDP/lpg/node5.html (1995).

[38] K. Jain, R. Sekar, User-level infrastructure for system call interposition: A platform for intrusion detection and confinement, in: Proceedings of the Network and Distributed Systems Security, 2000, pp. 19–34.

[39] M. Frigo, S. Jhonson, Discrete fourier transformation, Wolrd Wide Web, http://www.fftw.org (2003).

[40] M. Yarrow, Nas parallel benchmarks suite 2.3 - is, Tech. rep., NASA Ames Research Center, Moffett Field, CA (1995).

[41] P. Mucci, Llcbench (low-level characterization benchmarks) home page, World Wide Web, http://icl.cs.utk.edu/projects/llcbench/ (July 2000).

[42] Q. Snell, A. Mukler, J. Gustafson, Netpipe: Network protocol independent performance evaluator, in: Proceedings of the IASTED International Conference on Intelligent Information Management and Systems, 1996, http://www.scl.ameslab.gov/netpipe.

[43] K. Tan, R. Maxion, Determining the operational limits of an anomaly-based intrusion detector, IEEE Journal on Selected Areas in Communications, Special Issue on Design and Analysis Techniques for Security Assurance 21 (1) (2003) 96–110.

[44] R. Batchu, Y. S. Dandass, A. Skjellum, M. Beddhu, MPI/FT: A model-based approach to low-overhead fault-tolerant message-passing middleware, Cluster Computing 7 (4) (2004) 303–315.

[45] W. Gropp, E. Lusk, Fault tolerance in MPI programs, in: Proceedings of the Cluster Computing and Grid Systems Conference, 2002.

Author Biographies

German Florez-Larrahondo Dr. Florez-Larrahondo was born in Bogota, Colombia. He received a Bachelor degree in Systems Engineering from the Universidad Nacional de Colombia, as well as an MS and Ph.D. in Computer Science from Mississippi State University. His major research areas include machine learning, computer security and high-performance computing. Dr. Florez-Larrahondo currently works for Verari Systems Software designing middleware for high-performance applications.

Zhen Liu Dr. Liu was born in Beijing, China. He received his Bachelor degree in computer science at the University of Science and Technology of China in 2000, and his MS and Ph.D. in computer science from Mississippi State University in 2003 and 2005 respectively. His major research areas are computer security, intrusion detection, and machine learning. Dr. Liu currently works for Microsoft.

Yoginder S. Dandass Dr. Dandass is an Assistant Professor of Computer Science and Engineering at Mississippi State University. He received his Ph.D. in Computer Science from Mississippi State University in 2003 with an emphasis in high-performance computing. His research interests include high-performance computing, real-time systems, and reconfigurable computing. From 1997 to 2003, he was a research associate at Mississippi State. He also has over eight years of experience as an IT consultant prior to joining Mississippi State.

Susan M. Bridges Dr. Bridges is a professor of computer science and engineering at Mississippi State University. She received her B.S. in botany from the University of Arkansas in 1969, an M.S. in biology from the University of Mississippi in 1975, an M.C.S. in computer science from Mississippi State University in 1983, and her Ph.D. in computer science from the University of Alabama in Huntsville in 1989. Her research interests are in machine learning and data mining with applications in computer security and bioinformatics

Rayford Vaughn Dr. Vaughn received his Ph.D. from Kansas State University, Manhattan Kansas (USA) in 1988 in the area of computer science with an emphasis in computer

security. He joined Mississippi State University in 1997 as a faculty member. Prior to joining the University, he completed a twenty-six year career in the Army where he commanded the Army's largest software development organization and created the Pentagon agency that today centrally manages all Pentagon IT support. He is currently the Billie J. Ball Professor of Computer Science and Engineering and teaches and conducts research in the areas of Software Engineering and Information Security. Dr. Vaughn has over 70 publications to his credit and is an active contributor to software engineering and information security conferences and journals.

.

.