

Mining Health Models for Performance Monitoring of Services

Mithun Acharya¹ and Vamshidhar Kommineni²

¹*Department of Computer Science, North Carolina State University, Raleigh, NC, USA, 27695*

²*Microsoft Center for Software Excellence, One Microsoft Way, Redmond, WA, USA, 98052*
acharya@csc.ncsu.edu, vamshik@microsoft.com

Abstract—Online services such as search and live applications rely on large infrastructures in data centers, consisting of both stateless servers (e.g., web servers) and stateful servers (e.g., database servers). Acceptable performance of such infrastructures, and hence the availability of online services, rely on a very large number of parameters such as per-process resources and configurable system/application parameters. These parameters are available for collection as performance counters distributed across various machines, but services have had a hard time determining which performance counters to monitor and what thresholds to use for performance alarms in a production environment. In this paper, we present a novel framework called *PerfAnalyzer*, a storage-efficient and pro-active performance monitoring framework for correlating service *health* with performance counters. *PerfAnalyzer* automatically infers and builds health models for any service by running the standard suite of pre-deployment tests for the service and data mining the resulting performance counter data-set. A filtered set of performance counters and thresholds of alarms are produced by our framework. The health model inferred by our framework can then be used to detect performance degradation and collect detailed data for root-cause analysis in a production environment. We have applied *PerfAnalyzer* on five simple stress scenarios – CPU, memory, I/O, disk, and network, and two real system – Microsoft’s SQL Server 2005 and IIS 7.0 Web Server, with promising results.

I. INTRODUCTION

The Web has forever changed the traditional ways in which software is built and delivered to customers. With online models, business can be realized as a suite of services offered to the customers over the Internet. Services such as search, mail, storage, auction, and online stores are flourishing multi-billion dollar online business ventures today. Business can be conducted anywhere, anytime with millions and billions of transactions happening every day. Reliable infrastructure support is a must for such online ventures to survive. Online services such as search and live applications rely on large infrastructures in data-centers, consisting of both stateless servers (e.g., web servers) and stateful servers (e.g., database servers). Server downtimes cost companies billions of dollars in losses every year. Any downtime more than 0.1% is deemed unacceptable.

Performance testing [W01] can evaluate how online applications (in this paper, we use the term *application* and *service* to mean the same thing) respond to different inputs

and stress levels. However, performance testing can only be used to verify with some confidence that an application is able to perform under expected and peak load conditions. Performance testing establishes safe stress limits for applications and provides guidelines for various configurable parameters on which the application performance might depend on. These guidelines [W01] are usually set by experts, who are either developers of the system or have thorough experience in observing and understanding various complicated system behaviors. The premise of this paper is to address whether we can design an *expert system*, which aids system administrators not familiar with the system in outlining performance guidelines.

Data center and server farms span hundreds of thousands of machines, often geographically distributed. Even when such huge systems are thoroughly tested for performance prior to deployment, real life scenarios can cause unexpected failures, undiagnosable performance problems, and slowdowns. In the face of such events, it is often very difficult to pin point the root cause with performance testing guidelines. Along with the performance testing guidelines, we need a *pro-active, real-time* performance monitoring framework to predict server downtimes in production environments. Such a framework should help pin pointing the root cause of failures or undiagnosable performance problems. The real-time monitoring framework should incur *minimal storage and processor overhead*, almost running *transparently* with the monitored application. It is also important for such a system to have a *very low false positive and false negative rate*.

With company profits tightly tied to the acceptable and continual performance of huge server infrastructures that host online applications, it would be beneficial to design a generic real-time monitoring framework, *independent of applications*. Such a real-time framework also facilitates *dynamic capacity planning* and *performance tuning*. Dynamic capacity planning guides smart and dynamic distribution of resources such as memory, bandwidth, and processing power across the infrastructure for desired performance. Dynamic performance tuning aids real-time adjustments of resource and application parameters for optimum performance. Server failures are not always

because of resource starvation. Performance problems can also surface due to bad application design that does not correctly utilize the available resources at the disposal. By observing the right parameters, the monitoring framework can provide guidelines for application redesign to alleviate performance problems caused by improper resource utilization.

Processor	% Total processor time, % Total privileged time, % Total interrupt time
System	Context switches per second, Processor queue length
Memory	Available Mbytes, Pages per sec, Cache faults per sec
Process	Page faults per sec, and for each monitored process – working set, private bytes, and handle count
Network Interface	Bytes received/sec, Bytes sent/sec, Packets received discarded, Packets outbound discarded
.NET CLR Memory	Aspnet_wp counters for %Time in GC, #Bytes in all heaps, # of pinned objects, Large object heap size
.NET CLR Exceptions	Aspnet_wp counter for # Exceptions thrown per sec
.NET CLR Locks and Threads	aspnet_wp counters for Contention rate per second, Current queue length
.NET CLR Data	SqlClient: Current # connection pools, SqlClient: Current # pooled connections

Figure 1: System and .NET counters for Web Server

With company profits tightly tied to the acceptable and continual performance of huge server infrastructures that host online applications, it would be beneficial to design a generic real-time monitoring framework, *independent of applications*. Such a real-time framework also facilitates *dynamic capacity planning* and *performance tuning*. Dynamic capacity planning guides smart and dynamic distribution of resources such as memory, bandwidth, and processing power across the infrastructure for desired performance. Dynamic performance tuning aids real-time adjustments of resource and application parameters for optimum performance. Server failures are not always because of resource starvation. Performance problems can also surface due to bad application design that does not correctly utilize the available resources at the disposal. By observing the right parameters, the monitoring framework can provide guidelines for application redesign to alleviate performance problems caused by improper resource utilization.

A. Performance Counters

Acceptable performance of server infrastructures, and hence the availability of online services, rely on a very large number of parameters such as per-process resource utilization and configurable system/application parameters. These parameters are available as performance counters distributed across various machines. Performance counters can monitor system components such as processors, memory, network, disk, and I/O and publish performance-related data. Apart from system and OS counters, an application can publish a specific set of counters for monitoring. User applications can also add custom performance counters or create counters dynamically during run-time, to observe select system/application behaviors. Figure 1 shows the expert-specified [W01], web server-specific counters, which might help identify potential web server bottlenecks during run-time. Due to space constraints, only a subset of counters is displayed here. Based on their experience and battery of pre-deployment performance tests, experts specify thresholds (not shown in the figure) for each counter, which determine healthy web server operation.

In this paper, we present **PerfAnalyzer**, a storage-efficient and pro-active performance monitoring framework based on performance counters. Even application experts have had a hard time determining which performance counters to monitor and what thresholds to use for performance alarms in a production environment. The general problem of designing an expert system to predict performance problems, with no user input whatsoever, is hard, and we make a few *assumptions* to simplify the problem while still being realistic. Prior to the deployment of an application in production environments, a stress tester tests the application in-house with different loads (ideally, as varied as possible, being representative of loads seen at application production environments), while recording different (ideally, all) performance counters. We assume that the stress tester can roughly distinguish between good and bad application *health* (for example, a high response time in a web service could be bad). The stress tester, however, cannot tell which performance counters are the predictors of application health and how. To this end, our framework analyzes the collected performance counter data and builds health models with relevant counters (ideally, a very small set of counters). These counters and the corresponding health model can be used to predict health of the application when deployed in production environments. The small set of counters can aid the system builders in root cause analysis, performance tuning, and capacity planning. Our framework complements the performance testing phase during which observations on performance counter data are done. The observations are used to construct a *health model* to pro-actively monitor the application at production environments, transparently. PerfAnalyzer is independent

of the application it analyzes and bases its inference solely on the observed performance counter data-set. On a very high level, PerfAnalyzer collects vast amounts of performance counter data during the performance testing phase. PerfAnalyzer applies techniques from statistics and data mining on this data set to construct a health model for the application. The health model is then deployed with the application, for real-time performance monitoring. The health model is used to predict performance degradation and collect detailed data for root cause analysis in production environments.

B. Contributions

Our framework applies rigorous data mining algorithms on performance counter data-set, for building real-time health model for services. In summary, this paper makes the following main contributions:

Experimental methodology to collect performance counter data: We outline various steps required to perform stress tests during the pre-deployment performance testing phase to collect useful performance counter data. The performance counter data-set is then used by our framework to mine health models. We propose three *data-categorization* techniques required to prepare the data set for further analysis.

Mining health models using performance counters: We apply techniques from statistics and data mining on performance counter data-set collected during the performance testing phase to automatically build storage-efficient and pro-active health models for services. PerfAnalyzer is designed to be independent of the application it analyzes.

Implementation: We have implemented our framework in a tool called PerfAnalyzer, which can be used to build health models for any service by analyzing performance counter data-set. Extensive experiments were conducted to determine what combinations of algorithms and thresholds yield the best health model.

Evaluation: We used PerfAnalyzer to automatically build health models for five simple stress scenarios – CPU, Memory, Network, Disk, and I/O, and two real scenarios - SQL Server 2005, and IIS 7.0 Web Server, with promising results.

The remainder of this paper is structured as follows. Section II starts with a synthetic and simplistic example that motivates our framework. Section III describes our framework in detail and introduces the various components of PerfAnalyzer, along with the experiment methodology. Section IV presents the implementation details and

evaluation results. Section V discusses related work. Finally, we conclude in Section VI and discuss future work.

II. EXAMPLE

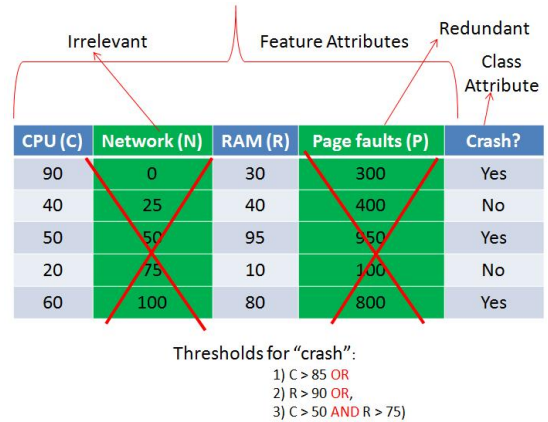


Figure 2: Crash data and crash model

In this section, we present a simple, yet illustrative example to demonstrate how PerfAnalyzer derives the health model from the observed performance counter data-set. Assume a user plays a game of chess against his desktop computer and finds that the chess application crashes unpredictably. To investigate the crash problem, the user decides to observe the four well-known performance counters available on his computer. The user observes %CPU utilization (**C**), %network utilization (**N**), %RAM utilization (**R**), and page faults per second (**P**) at five different times – at times when the chess application is running fine and at times when the chess application crashes. On a Windows machine, these counters are available through Microsoft *Windows Task Manager*, often invoked through the *ctrl-alt-del* key sequence. For each instance, the user also records the crash status (**crash**), i.e., if the chess application crashed (**yes**) or not (**no**). The four attributes, **C**, **N**, **R**, and **P** are called *feature attributes* and the crash status, **crash**, is called the *class attribute*. The collected data is shown in Figure 2.

The goal of PerfAnalyzer is to derive the “crash model” for the chess application from the collected crash data. The crash model can be used to infer (or predict) the crash status at any arbitrary time when the values for the feature attributes are known. In other words, the problem is to infer (or predict) the unknown class attribute **crash** (**yes** or **no**) for arbitrary values of the feature attributes (**C**, **N**, **R**, and **P**). The inference is based on the previously collected crash data. The four feature attributes are representative of different performance counters available on any system. The crash model is analogous to application health model. PerfAnalyzer derives the crash model from the synthetic data-set in two phases. In the first phase, it identifies the

irrelevant and *redundant* attributes. This step is called *feature selection*. Irrelevant attributes do not have any correlation with the class attribute. An attribute is redundant, if another attribute can accurately predict the class attribute as well as or better than the redundant attribute. In our example, PerfAnalyzer determines that the network utilization (**N**) is an irrelevant attribute (no correlation with the crash status; probably the chess application does not use the network at all). Furthermore, PerfAnalyzer determines that attribute **P** (page faults per second) is redundant. %RAM utilization (**R**) can be used in lieu of page faults per second (**P**) to predict the crash status (in our synthetic example, it turns out that **P** is exactly 10 times **R**; page faults per second is expected to increase with RAM utilization). PerfAnalyzer employs various feature selection algorithms to identify redundant and irrelevant feature attributes. Once irrelevant and redundant attributes are pruned, PerfAnalyzer constructs the crash model by identifying thresholds for the remaining attributes. This constitutes the second phase, in which various regression algorithms [A02] are used. For the synthetic crash data shown in Figure 2, PerfAnalyzer derives the following crash model:

$$\text{crash} = ((C > 85) \mid (R > 90) \mid ((C > 50) \wedge (R > 75)))? \text{ yes : no}$$

Challenges: The example motivates the idea of mining health models from performance counter data-set. However, there are many issues, not obvious in the motivating example, and these issues shall be addressed throughout the paper. **(1)** There were only four parameters in the synthetic data-set and we could derive the crash model manually by inspection. Furthermore, as shown in Figure 1, for web server, performance can also be affected by related applications such as SQL Server. Even a system expert might easily miss out on such interactions. Manually identifying thresholds becomes impossible if desired system behavior depends on thousands of performance counters with complex interactions. Furthermore, in our example, the crash model derived from only five observations will be very inaccurate. **(2)** Performance counters often do not share linear relation with the application health, and complex regression equations are required to capture the correlation. **(3)** In our example, the crash status was a simple *dichotomous* variable with only two values. The health index for real applications can be much finer with possibly no apparent ordering between different indices. **(4)** Since the health model depends heavily on the performance counter data-set, a systematic methodology to conduct performance tests at various health levels should be in place. **(5)** Finally, it is not at all obvious which feature selection and regression algorithms (or combinations) result in the best health model. Also, for each algorithm used, we conducted extensive experiments to estimate the correct *cut-off* values (for a given algorithm, counters are retained or discarded based on the *cut-off* value

for that algorithm; see Section III.D). In the next section, we present our framework and show how these problems are addressed.

III. FRAMEWORK

In this section, we formalize the several notions introduced in the previous sections. We formalize the problem of mining health models from performance counter data-set. Next, we describe the monitoring framework needed to capture the counter data-set. After outlining the experiment methodology required for collecting the counter data-set, we present the two main components of our framework – first-pass filters and second-pass filters.

A. Problem Statement

Let $S = \{c_1, c_2, c_3, \dots, c_n\}$ be a set of n ($= |S|$) user-determined performance counters (feature attributes), which can be observed during application run-time. For our experiments, we choose S to be *all* the counters available on the machine running our target application (we sometimes use the word *application* to mean target application). PerfAnalyzer monitors counters from a single machine on which the target application runs. However, it is a simple generalization to gather performance counters from multiple machines. Our framework makes no assumption on the number of performance counters, $|S|$, or which machine they are from. This design makes PerfAnalyzer generic and independent of the number of machines and the service to be monitored. At any particular instance of time t , S_t , a *snapshot* of S is the recorded value for each counter in S at instance t . For example, $\{c_1, c_2, c_3, \dots, c_n\} = \{123, 10989, 531, \dots, 76\}$ at some time t is a snapshot of set S . Generally, a snapshot of any counter set is the recorded value for each counter in that set at some instance of time. Also, recorded snapshots of a subset of S (say, S') from the data-set D is the set of recorded values for counters in S' . Snapshots of S are recorded every Δ seconds. We identify two phases during which counter snapshots can be collected. Counter snapshots can be collected either during the pre-deployment performance testing phase, or in real-time, after the application is rolled out at the production environment.

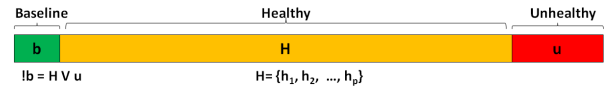


Figure 3: Graphical view of health indices

k snapshots are recorded during pre-deployment performance tests, starting from time $t = 0$, yielding snapshots $S_0, S_{\Delta}, S_{2\Delta}, \dots, S_{(k-1)\Delta}$. For simplicity, we omit Δ in the subscript and represent the snapshots as a set $T = \{S_0, S_1, S_2, \dots, S_{k-1}\}$. T denotes counter snapshots collected

during pre-deployment performance tests. We use \mathbf{R} to denote counter snapshots recorded during application run-time at production time. Performance counter data is collected at various application *health levels*. In this paper, we assume that the user can at least always distinguish between the three application health levels - baseline (\mathbf{b}), healthy (\mathbf{h}), and unhealthy (\mathbf{u}), described next.

An application is said to run at baseline health level \mathbf{b} , when it is just “ON”. For example, SQL Server is said to run at baseline health level, when the SQL server is up and ready, but not processing any requests or queries. The methodology to conduct baseline experiments is presented in Section III.C. An application is said to run at some healthy level \mathbf{h}_j , if the observed (by the user) application performance is healthy and acceptable. Finally, when the performance levels are unacceptable, the application is said to run at unhealthy level \mathbf{u} . Healthy levels may be partitioned into \mathbf{p} finer levels, $\mathbf{H} = \{\mathbf{h}_1, \mathbf{h}_2, \dots, \mathbf{h}_p\}$. Note the distinction between the terms *health levels* and *healthy levels*. \mathbf{b} , \mathbf{u} , and any index in \mathbf{H} are different health levels. Indices in \mathbf{H} , namely, $\mathbf{h}_1, \mathbf{h}_2, \dots, \mathbf{h}_p$ are different healthy levels. These finer indices are either ordered or unordered. That is, if the \mathbf{H} indices are ordered, then we may assume that $\mathbf{h}_1 < \mathbf{h}_2 < \mathbf{h}_3 < \dots < \mathbf{h}_p$. An application with healthy index \mathbf{h}_a is healthier than the same application with healthy index \mathbf{h}_b iff $\mathbf{h}_a < \mathbf{h}_b$, $1 \leq a, b \leq p$ (lower index, better health). If \mathbf{H} indices are unordered, then given any two indices \mathbf{h}_a and \mathbf{h}_b , $1 \leq a, b \leq p$, it is not possible to determine if $\mathbf{h}_a < \mathbf{h}_b$, i.e., the only conclusion that can be safely derived is that the two healthy indices probably indicate different healthy levels. In Section III.C, we propose different techniques for handling ordered and unordered indices. Furthermore, the algorithms for second-pass filters (See Section III.E) depend on the type of healthy indices. We club all \mathbf{H} indices and \mathbf{u} index under one health index, $\mathbf{!b}$ (not baseline). The different health levels are shown graphically in Figure 3. Clearly, $\mathbf{!b} = \mathbf{H} \cup \mathbf{u}$. If \mathbf{I} is the set of all indices, then $\mathbf{I} = \{\mathbf{b}, \mathbf{H}, \mathbf{u}\} = \{\mathbf{b}, \mathbf{h}_1, \mathbf{h}_2, \dots, \mathbf{h}_p, \mathbf{u}\} = \{\mathbf{b}, \mathbf{!b}\}$. We use set \mathbf{B} to denote the indices \mathbf{b} and $\mathbf{!b}$. For each snapshot, we append a health index as observed by the user. We term the set $\mathbf{D} = \{(\mathbf{S}_0, \mathbf{b}_0, \mathbf{i}_0), (\mathbf{S}_1, \mathbf{b}_1, \mathbf{i}_1), \dots, (\mathbf{S}_{k-1}, \mathbf{b}_{k-1}, \mathbf{i}_{k-1})\}$, $\mathbf{i}_j \in \mathbf{H} \cup \mathbf{u}$, $0 \leq j < k$, $\mathbf{b}_j \in \mathbf{B}$ (either \mathbf{b} or $\mathbf{!b}$), as the performance counter data-set. For all $(\mathbf{S}_j, \mathbf{b}_j, \mathbf{i}_j)$ in \mathbf{D} , where $\mathbf{b}_j = \mathbf{b}$, $\mathbf{i}_j = \mathbf{NULL}$, i.e., no healthy/unhealthy indices for baseline data. \mathbf{D} is stored as a $[k \times (|\mathbf{S}| + 2)]$ matrix, with $|\mathbf{S}|$ feature attributes, two indices, and k counter snapshots. PerfAnalyzer mines the health model by analyzing \mathbf{D} and computing $\mathbf{S2}$. To compute $\mathbf{S2}$, an intermediate set $\mathbf{S1}$ is computed initially. $\mathbf{S1}$, a subset of \mathbf{S} , is an intermediate set which contains all the counters in \mathbf{S} except those that are found irrelevant or redundant by PerfAnalyzer. To compute $\mathbf{S1}$, the counters in \mathbf{S} (feature attributes) are correlated against the class attribute \mathbf{B} . PerfAnalyzer employs various feature selection algorithms to compute $\mathbf{S1}$ from \mathbf{S} and \mathbf{D} (first pass; see Section III.D).

$\mathbf{S2}$, a subset of $\mathbf{S1}$, is the final set of counters used in building the health model. The reason for computing $\mathbf{S2}$ from $\mathbf{S1}$ instead of \mathbf{S} directly is given in Section III.D. To compute $\mathbf{S2}$, and the health model, the counters in $\mathbf{S1}$ are regressed against the class attribute \mathbf{H} (second pass; see Section III.E). To summarize, the problem is to compute $\mathbf{S2}$, and a health model based on $\mathbf{S2}$, given \mathbf{S} (user-provided) and \mathbf{D} (recorded by the monitoring agent). The application is then deployed with the mined health model.

Mapping the notations to the example in Section II, $\mathbf{S} = \{\mathbf{C}, \mathbf{N}, \mathbf{R}, \mathbf{P}\}$ and $\mathbf{I} = \{\mathbf{yes}, \mathbf{no}\}$. For our example, $\mathbf{S1} = \mathbf{S2} = \{\mathbf{C}, \mathbf{R}\}$. Many issues already outlined in Section II such as the need for baseline experiments are not immediately obvious from our example. Finally, for our simple synthetic example, $|\mathbf{S}| > |\mathbf{S1}| = |\mathbf{S2}|$. But, generally, for real applications, $|\mathbf{S}| \gg |\mathbf{S1}| > |\mathbf{S2}|$, as confirmed by our evaluation in Section IV. In the next few sections, we describe the various components of our framework along with the experiment methodology.

B. Monitoring Agent

The monitoring agent (in short, agent) records counter snapshots during the pre-deployment performance-test phase and at the application production site, post-deployment. Counter snapshots are recorded every Δ seconds, when the application is running. The agent (monitoring component of PerfAnalyzer) and the target application run on the same machine. Remaining components of PerfAnalyzer run on a remote machine and interact with the agent. Henceforth, we term the machine on which the target application runs as the *testbed machine* and the machine on which PerfAnalyzer resides as the *remote machine*. The counter set to be monitored (\mathbf{S}) on the testbed machine and Δ can be specified in a configuration file. For our experiments, we choose \mathbf{S} to be *all* the counters available on the testbed machine. The agent initially probes the testbed machine to determine what counters are published and it includes all the published counters in the set \mathbf{S} . PerfAnalyzer monitors counters from a single testbed machine. However, it is a simple generalization to gather performance counters from multiple testbed machines with the concepts of *roles* and *aggregation* discussed in Section VI. The agent runs along with the application and sends the counter snapshot information over the network to the remote machine. Furthermore, many processes, mostly from the Operating System, not related to the target application, still run on the testbed machine, possibly skewing the recorded counter data. These effects, as we show in Section III.C, are neutralized by conducting baseline experiments and computing an intermediate set $\mathbf{S1}$, based on the baseline data. Other configurable parameters for the agent include the ramp-up time, r , and the moving average window size, m . Counter snapshots are not recorded for the first r

seconds during the startup or initialization phase of the target application. Furthermore, recorded counter snapshots are averaged every m time periods to account for possible local spikes. The agent dumps the recorded counter snapshots into a database server on the remote machine. PerfAnalyzer employs SQL Server 2005 to store and process performance counter data-set.

C. Experiment Methodology

In this section, we outline the various steps required to collect \mathbf{D} , the performance counter data-set. Before proceeding to the details of the methodology, we present a high-level overview of our framework, shown in Figure 4. The agent takes \mathbf{S} , specified in the configuration file by the user, as input. Stress experiments are conducted on the testbed machine. The agent monitors the target application on the testbed machine. The counter snapshots are dumped on a database, hosted remotely. PerfAnalyzer employs SQL Server 2005 to store and process performance counter data-set. A first-pass filter takes \mathbf{S} and the recorded counter snapshots of \mathbf{S} on the database (\mathbf{T}) as input and generates $\mathbf{S1}$. Finally, the second-pass filters take $\mathbf{S1}$ and recorded snapshots of $\mathbf{S1}$ from the database to generate $\mathbf{S2}$, and a health model for the target application. To collect the performance counter data-set \mathbf{D} , we perform experiments in two stages – *baseline* and *stressed*. No stress is applied on the application in the baseline stage. Varying stress levels are applied on the application in the stressed stage. We next describe the procedure required to conduct baseline and stressed experiments and the need for them.

Baseline experiments: Many processes, mostly from the Operating System, and the agent run on the testbed machine along with the application. It is important to take into account the effect these processes have on the monitored performance counters. In other words, the final filtered set should include only those counters that are good predictors of application health. To neutralize the effect of unrelated processes, we perform experiments in two stages – *baseline* and *stressed*. We notice that the processes that are not related to the target application are running during both stages. Hence the unrelated processes more or less have the same effect on the counters during baseline and stressed stages. A first-pass filter makes use of this fact to get rid of those counters that are not at all related to the application stress. In the baseline stage, the application is just up and ready without doing any processing. In the stressed stage (explained next), varying levels of stress are applied on the application. Each snapshot of \mathbf{S} recorded during the baseline stage is appended with an index \mathbf{b} to indicate *baseline*. For all $(\mathbf{S}_j, \mathbf{b}_j, \mathbf{i}_j)$ in \mathbf{D} , where $\mathbf{b}_j = \mathbf{b}$, we have $\mathbf{i}_j = \text{NULL}$, as no stress is applied on the application at this stage.

Stress experiments: In this stage, the user applies varying levels of stress to the application, often trying to *break* the application at unhealthy stress levels. Each snapshot of \mathbf{S} recorded during the stressed stage is appended with an index $\mathbf{!b}$ to indicate *not baseline*. Once all the data is indexed with either \mathbf{b} or $\mathbf{!b}$, the data-set is ready for first-pass filtering. Each snapshot of \mathbf{S} recorded during the stressed stage is also appended with a health index \mathbf{i} , \mathbf{iCH} (healthy levels) or \mathbf{u} (unhealthy levels). PerfAnalyzer assumes that the user can at least always distinguish between baseline, healthy, and unhealthy levels for the application. However, when experiments are conducted at two different healthy levels, \mathbf{h}_a and \mathbf{h}_b , PerfAnalyzer only assumes that \mathbf{h}_a and \mathbf{h}_b are different and does not assume any ordering between them. However, if the ordering is obvious to the user, she explicitly indicates it to PerfAnalyzer. To recall the definitions from Section III.A, if the \mathbf{H} indices are ordered, then we may assume that $\mathbf{h}_1 < \mathbf{h}_2 < \mathbf{h}_3 < \dots < \mathbf{h}_p$. An application with healthy index \mathbf{h}_a is healthier than the same application with healthy index \mathbf{h}_b iff $\mathbf{h}_a < \mathbf{h}_b$, $1 \leq \mathbf{a}, \mathbf{b} \leq \mathbf{p}$. If \mathbf{H} indices are unordered, then given any two indices \mathbf{h}_a and \mathbf{h}_b , $1 \leq \mathbf{a}, \mathbf{b} \leq \mathbf{p}$, it is not possible to determine if $\mathbf{h}_a < \mathbf{h}_b$, i.e., the only conclusion that can be safely derived is that the two indices probably indicate different healthy levels. If \mathbf{H} indices are ordered, then \mathbf{p} denotes the number of different healthy levels user can distinguish. If \mathbf{H} indices are unordered, then \mathbf{p} equals number of distinct sets of experiments conducted at different unknown stress levels. Snapshots from a single set of experiments or unknown stress level are assigned to a distinct bucket in $[\mathbf{1}, \mathbf{p}]$. Ordered and unordered indices lead to three types of data-categorization – *continuous-ordered*, *rank-ordered*, and *tag-ordered*, explained next. Data categorization is a necessary step prior to second-pass filtering.

Data Categorization: Ordered healthy indices lead to two types of data categorization – *rank-ordered* and *continuous-ordered*. For rank-ordered healthy indices, the absolute difference among indices does not matter. In other words, $|\mathbf{h}_a - \mathbf{h}_b|$ is immaterial for any \mathbf{a}, \mathbf{b} in $[\mathbf{1}, \mathbf{p}]$. Only the ordering matters, and not the absolute difference. The indices in such cases are known as *ranks*. As an example, for a bandwidth intensive application, though the user can tell that a net bandwidth of 100Mbps is better than 1Mbps, she cannot comment on how much better. For continuous-ordered healthy indices, the user precisely knows the absolute difference between any two indices. In other words, $|\mathbf{h}_a - \mathbf{h}_b|$ matters and is precisely known to the user, for any \mathbf{a}, \mathbf{b} in $[\mathbf{1}, \mathbf{p}]$. Continuous-ordered healthy indices are used when an external visible effect, \mathbf{E} , is available all the time during baseline and stressed experiments. The observed values for \mathbf{E} are used as healthy indices. As \mathbf{E} is continuous, we have $\mathbf{p}=\infty$. Some examples of external visible effect include transactions per second (TPS), latency, and throughput. These values are treated as

continuous variables. In some cases such as data-centers, an externally visible effect might not be readily available. Also rank-ordering might not be possible. In such cases, we use *tag-unordered* data categorization. If the healthy indices are unordered, then the indices are *nominal* and PerfAnalyzer treats different indices as *tags*. The regression algorithms used by the second-pass filters depend on whether the healthy indices are continuous-ordered, rank-ordered, or tag-unordered. The collected performance counter data-set is explicitly marked with the appropriate data-category, before second-pass filters are applied. We next describe the first and second-pass filters, which analyze the data collected by the monitoring agent.

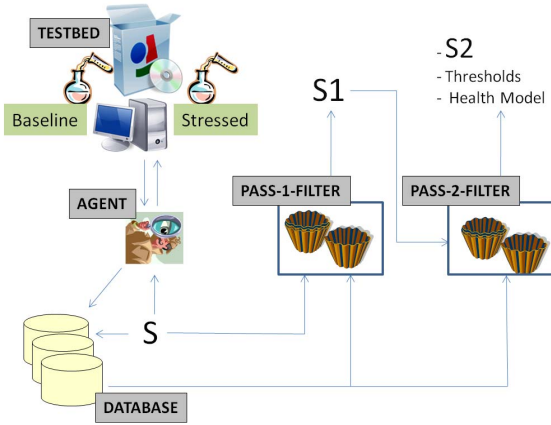


Figure 4: Framework for mining health model from performance counter data-set

D. First-pass Filters

The counter data-set collected during baseline and stressed stages are ready for filtering after data-categorization. First-pass filters operate on snapshots of \mathbf{S} from \mathbf{D} . The effect of processes not related to the target application is more or less the same during baseline and stressed stages. The key goal of the first-pass is to discard all counters that are not at all correlated with the application health. These uncorrelated counters follow similar patterns in baseline and stressed stages. To identify such counters, the first-pass filters analyze the relationship between individual counters in \mathbf{S} with \mathbf{B} (\mathbf{b} or $!\mathbf{b}$), using different algorithms listed in Figure 5. Counters in \mathbf{S} are called *feature attributes*, and \mathbf{B} is called the *class attribute*. Useless (irrelevant or redundant) counters in \mathbf{S} are removed to produce a cleaner set of counters in $\mathbf{S1}$. For any snapshot of \mathbf{S} in \mathbf{T} (counter snapshots recorded during pre-deployment tests), PerfAnalyzer can automatically determine (with certain confidence \mathbf{d}), if the application health index is \mathbf{b} or $!\mathbf{b}$, using counters in $\mathbf{S1}$. In data mining terms, this process is called *data-cleaning* or *feature-selection*. The first-pass filter computes a confidence value \mathbf{d}_{ci} for each counter $\mathbf{c}_i \in \mathbf{S}$, $\mathbf{i} \in [1, \mathbf{n}]$. All counters $\mathbf{c}_i \in \mathbf{S}$, $\mathbf{i} \in [1,$

$\mathbf{n}]$, with $\mathbf{d}_{ci} < \mathbf{f}$, a fixed estimated *cut-off* value for the filter, are discarded. The cut-off value \mathbf{f} differs for each filter. We conducted extensive experiments to estimate the best \mathbf{f} value for each filter used in the first pass. We have implemented different feature selection algorithms in PerfAnalyzer (see Figure 5) to determine which algorithms produce the best result. A discussion about the various first pass filters employed by PerfAnalyzer is given next.

Algorithm	Key Idea – Candidate filters – Cut-off (\mathbf{f})
Machine Learning Algorithms	Apply ML algorithms directly on \mathbf{S} without <i>data-cleaning</i> - Decision Tree, Naïve Bayes, and Dichotomous Logistic Regression
Correlation Algorithms	Capture linear correlation between counters in \mathbf{S} and \mathbf{B} – Point biserial correlation – 0.6
Entropy	Capture <i>disorder</i> information for a counter \mathbf{C} in \mathbf{S} , independent of \mathbf{B} – 0.1
Information Gain	How much <i>information</i> can be gained about class attribute \mathbf{B} by observing a counter \mathbf{c} in \mathbf{S} ? – IGSU – 0.2

Figure 5: First-pass filters

Machine Learning (ML) algorithms: We directly used three ML algorithms [B07, W01] – Decision Tree (DT), Naïve Bayes (NB), and Dichotomous Logistic Regression (DLR), over \mathbf{D} , to produce health models directly from \mathbf{S} (without generating intermediate set $\mathbf{S1}$). DT algorithm failed when some feature attributes appeared perfect and terminated pre-maturely. It did not help us to use DT in the first pass. NB assumes that the feature attributes are *independent*. The algorithm does not take into account the dependencies that may exist. Hence it was not suitable for analyzing performance counter data-set \mathbf{D} , in which several dependencies may exist. Dichotomous Logistic Regression (DLR) [A02] predicts the class attribute \mathbf{B} which can have only two values – \mathbf{b} or $!\mathbf{b}$. The feature attributes may be continuous, discrete, dichotomous, or a mix of any of these. *Discriminant analysis* [D84] can also be used to predict dichotomous class attribute \mathbf{B} . However, discriminant analysis can only be used with continuous, independent variables. Since we cannot know the type of thousands of performance counters (feature attributes), we did not include discriminant analysis among our filters. DLR also ranks the relative importance of each feature attribute. However, when we applied DLR on \mathbf{S} directly, the resulting set of counters and health model were very poor. It helped us to employ simpler filters (explained next) before using LR filters. LR filters performed very well when a clean set of counters were presented to them ($\mathbf{S1}$ instead of \mathbf{S}). This was our primary motivation to employ LR filters for second pass (See Section III.E). When the class attribute is *ordinal* or *ranked*, we use Ordinal Logistic Regression (OLR). When the class attribute is *nominal*, we use Multinomial

Logistic Regression (MLR). OLR and MLR filters are used in the second pass after data-cleaning by simpler first pass filters, explained next.

Correlation Algorithms: In statistics, *correlation* [CCWA03] or *correlation co-efficient* indicates the measure of linearity between two random variables. While various correlation algorithms exist depending on the type of feature attributes and class attribute, Point Biserial Correlation (PBC) is used when the class attribute is dichotomous. In our case, the class attribute \mathbf{B} is dichotomous and hence we included PBC among our first pass filters. The equation for computing PBC between a feature attribute \mathbf{X} and the class attribute \mathbf{B} is $\mathbf{pbc} = ((\mathbf{M}_1 - \mathbf{M}_0) / \sigma) \sqrt{\mathbf{n}_1 \mathbf{n}_0 / (\mathbf{n}(\mathbf{n}-1))}$, where \mathbf{pbc} is the PBC coefficient, \mathbf{M}_1 is the mean value of the variable \mathbf{X} for all snapshots with $\mathbf{B} = \mathbf{b}$, \mathbf{M}_0 is the mean value of \mathbf{X} for all snapshots with $\mathbf{B} = !\mathbf{b}$, and σ is the standard deviation for all data points taken collectively. Further, \mathbf{n}_1 is the number of data points with $\mathbf{B} = \mathbf{b}$, \mathbf{n}_0 is the number of data points with $\mathbf{B} = !\mathbf{b}$, and \mathbf{n} is the total sample size. We conducted extensive experiments to find the cut-off value \mathbf{f}_{PBC} for the PBC filter. We found that $\mathbf{f}_{\text{PBC}} = 0.6$ gave us the best results. We could conclude that all counters in \mathbf{S} with confidence $\mathbf{d} > \mathbf{f}_{\text{PBC}} = 0.6$ had a strong linear correlation with the class attribute \mathbf{B} . These counters were definitely included in $\mathbf{S1}$. However, nothing could be said about the counters with $\mathbf{d} < \mathbf{f}_{\text{PBC}} = 0.6$. Absence of linear correlation does not rule out the possibility of non-linear correlation or other relationships between the counters and the class attribute \mathbf{B} . Other first pass filters based on *entropy* and *information gain* were required for further data-cleaning. These filters are explained next.

Entropy Algorithms: Entropy based filter analyzes the feature attributes in \mathbf{S} independent of the class attribute \mathbf{B} . The entropy filter measures the disorder among feature attributes. For a given feature attribute \mathbf{X} , the entropy $\mathbf{E} = \mathbf{H}(\mathbf{X})$ is measured as $\mathbf{H}(\mathbf{X}) = -\sum_{\mathbf{x} \in \mathbf{X}} \mathbf{p}(\mathbf{x}) \log \mathbf{p}(\mathbf{x})$, where $\mathbf{p}(\mathbf{x})$ is the probability that the value of $\mathbf{X} = \mathbf{x}$. For a given feature attribute, let \mathbf{E}_B and \mathbf{E}_S be the entropy for baseline and non-baseline snapshots respectively. From our experiments, we determined that all the counters with $\mathbf{E}_S < \mathbf{f}_{\text{Entropy}} = 0.1$ are useless and should be discarded. These counters do not change much during stressed period. We also computed $\delta = |\mathbf{E}_B - \mathbf{E}_S|$ for each counter. If δ for a counter was very high, we definitely selected the counter to be in $\mathbf{S1}$. Counters with high δ changed meaningfully between baseline and stressed stages of the application. The simple entropy filter was experimentally found to be very effective in reducing the counter size, and is recommended to be applied before any other filter.

Information Gain Algorithms: Information Gain (IG) filter computes how much more *information* is gained about the class attribute by knowing a feature attribute. IG

filter is used after using the Entropy filter. Unlike Entropy filter, IG filter analyzes the feature attributes in relation with the class attribute. Let \mathbf{X} be a feature attribute and \mathbf{Y} be a class attribute. Entropy of \mathbf{Y} before observing \mathbf{X} is $\mathbf{H}(\mathbf{Y})$. Entropy of \mathbf{Y} after observing \mathbf{X} is $\mathbf{H}(\mathbf{Y}|\mathbf{X})$, given by $\mathbf{H}(\mathbf{Y}|\mathbf{X}) = -\sum_{\mathbf{x} \in \mathbf{X}} \mathbf{p}(\mathbf{x}) \sum_{\mathbf{y} \in \mathbf{Y}} \mathbf{p}(\mathbf{y}|\mathbf{x}) \log \mathbf{p}(\mathbf{y}|\mathbf{x})$. To compute the Information gain, \mathbf{IG} , we use $\mathbf{IG} = \mathbf{H}(\mathbf{Y}) - \mathbf{H}(\mathbf{Y}|\mathbf{X})$. Unfortunately, \mathbf{IG} is biased in favor of features with more values, that is attributes with greater numbers of values will appear to gain more information than those with fewer values even if they are no more informative. Also, all \mathbf{IG} s should be normalized. Symmetrical uncertainty (\mathbf{SU}) compensates for \mathbf{IG} 's bias towards attributes with more values and normalizes its value to the range $[0, 1]$. To compute \mathbf{SU} , we use $\mathbf{SU} = 2 * \mathbf{IG} / (\mathbf{H}(\mathbf{Y}) + \mathbf{H}(\mathbf{X}))$. From our experiments, we found that the cut-off, $\mathbf{f}_{\text{IGSU}} = 0.2$, gave the best results for the filter using Information Gain with Symmetrical Uncertainty (IGSU) [H98].

E. Second-pass Filters

Second-pass filters operate on snapshots of $\mathbf{S1}$ from \mathbf{D} . First-pass filters correlate counters in \mathbf{S} with the class attribute \mathbf{B} to produce $\mathbf{S1}$. Second-pass filters correlate counters from $\mathbf{S1}$ with the class attribute \mathbf{H} , which represents the set of healthy/unhealthy indices. The purpose of the first-pass filter is to provide a *cleaner* set of data to the second-pass. First-pass filters do not take into consideration the interaction between different counters in \mathbf{S} . The key goals of second-pass filters are to produce the set $\mathbf{S2}$ to predict \mathbf{H} indices and various counter thresholds that define the health model. Thresholds of different counters in $\mathbf{S2}$ are embedded in the regression equations involving the different counters. PerfAnalyzer uses different regression algorithms based on data-categorization. The algorithms used are summarized in Figure 6.

For any snapshot of $\mathbf{S2}$ in \mathbf{R} (counter snapshots recorded post-deployment), PerfAnalyzer can automatically determine the application health index \mathbf{i} (with confidence \mathbf{d}), $\mathbf{iCH|u}$. Like the first-pass filters, each second-pass filter has a fixed cut-off value to discard counters. From our experiments, we found that $\mathbf{f}_{\text{regression}} = 0.5$ was a good cut-off value for OLR and MLR filters. The application is then deployed with the mined health model. The health model monitors only counters from $\mathbf{S2}$ at the application production environment to automatically compute the health index in real-time. Based on the estimated health index, predictions can be made on the general application health. For example, if the health deteriorates continuously within a certain time period, say ω , alarm should be raised. In the production environment, snapshots are taken every Δ seconds, but only the last ω snapshots are stored. The snapshots of $\mathbf{S2}$ recorded during the last ω time periods will provide useful hints for analyzing the root cause of failure

or health deterioration, if any. If moving average is computed every m time periods, then it is sufficient to store the last ω/m values for each counters in S_2 . Hence the health model incurs a constant memory complexity of $\mathcal{C}(|S_2| \times \omega/m)$, where \mathcal{C} is some implementation dependent constant.

IV. IMPLEMENTATION AND EVALUATION

We have implemented PerfAnalyzer in C#, on Microsoft CLR platform. The monitoring agent was developed internally at Microsoft. PerfAnalyzer employs SQL Server 2005 to store and process the counter data-set. SQL Server’s business intelligence and data mining algorithms were used programmatically in C# for accessing ML-based first-pass and second-pass LR filters. A screenshot of PerfAnalyzer is shown in Figure 7. Various configurable parameters such as data-categorization and filter cut-offs can be specified as menu options. PerfAnalyzer outputs S_1 , S_2 , and a health model. We have applied PerfAnalyzer on five simple stress scenarios – CPU, Memory, Disk, I/O, and network; and two real applications – SQL Server 2005 Enterprise edition and IIS Web Server 7.0. We next describe our evaluation criteria, validation approach, and results from seven case studies.

Evaluation Criteria: To evaluate PerfAnalyzer and the health model it mines, we used the following three qualitative and quantitative criteria. **(1)** How good are the counters in S_1 ? An expert inspected the counters selected in S_1 and the discarded counters in $(S - S_1)$. Are there any obvious counters missing (false negatives) in S_1 ? Are there any counters that are not expected to be there in S_1 ? (false positives) **(2)** How good are the counters in S_2 ? An expert inspected the counters in S_2 and $S_2 - S_1$. For the simple scenarios - CPU, memory, disk, I/O, and network, to some extent, we could manually predict the counters in S_1 and S_2 , and hence qualitatively evaluate the results from PerfAnalyzer. This was the primary reason why we chose five simple scenarios among our evaluation subjects. For example, for our CPU experiments, the *ProcessorTotal%UserTime* and *ProcessorTotal%ProcessorTime* counters were in S_1 and S_2 with highest confidence (d), as expected. **(3)** How much reduction is achieved in the counter size by using PerfAnalyzer? How well does the health model predict

health indices, post-deployment using counters in S_2 ? Our quantitative validation approach is explained next.

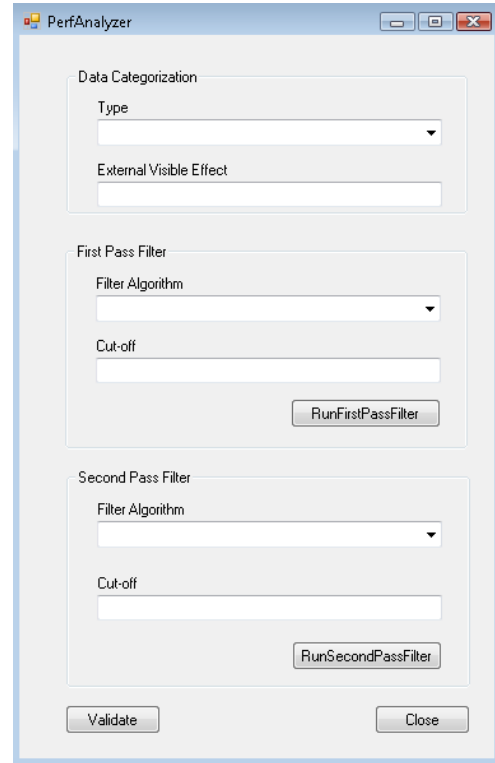


Figure 7: A screenshot of PerfAnalyzer

Validation: How well can the health model predict the health indices, post-deployment? To test this, we choose a random subset V (*test data-set*) from D (counter data-set). Each indexed-snapshot (S_j, b_j, i_j) in D with $b_j = !b$ for all j in $[0, (k-1)]$ is chosen to be in V with some user-determined probability q . Each (S_j, b_j, i_j) chosen from D to be in V is stripped of its indices. Also, all counters in $(S - S_2)$ are discarded. In other words, each non-baseline snapshot in D is chosen to be in V with some probability q . Only S_2 counters are retained in V . In fact, V roughly represents R , the counter snapshots of S_2 recorded during application run-time, post-deployment. The validation module takes V as input and predicts the health index for each S_2 snapshot in V , using the mined health model.

Assumption	Explanation	Technique Overview	Output
Continuous-Ordered	External effect (E) visible. H is the set of different observed values of E	Regression analysis on continuous E	Smaller set of counters (S2) that predict H and thresholds
Rank-ordered	Stress tester can rank experiments. H is the set of user-assigned ranks	Ordinal Logistic Regression on ordinal-rank H	Smaller set of counters (S2) that predict H and thresholds
Tag-unordered	stress-tester tags snapshots – H is the set of user-assigned nominal tags	Multinomial regression on multinomial-categorical H	Smaller set of counters (S2) that predict H

Figure 6: Second-pass filter algorithms

The inferred health index is compared with the corresponding actual health index available in **D** to determine the prediction accuracy. PerfAnalyzer excludes the test data-set **V** when building the health model. The health model is inferred from snapshots in (**D** – **V**). With the evaluation criteria and validation approach in place, we next describe our experience of using PerfAnalyzer on seven different subjects. For each subject, baseline experiments were done for about 2 hours. Experiments at each stress level (healthy and unhealthy) were performed for about 30 minutes. The monitoring agent, implemented as *agent.exe* on the monitoring machine collected counter snapshots from the target machine. Different stress tools [W02], developed internally at Microsoft, were used to apply stress to applications under test. As expected, most per-process counters for the stress tool process and *agent.exe* were picked.

CPU Stress: A CPU stressor [W02], *StressTool.exe*, was run on the dual-core target machine to vary CPU stress from 0 - 90%. Among other counters, per-process counters were monitored for *StressTool.exe* and *agent.exe*. A snapshot collected during CPU stress level of **a%** is assigned a lower rank (lower rank values indicate better health. For example, rank 2 is healthier than rank 6) than the snapshot collected during CPU stress level of **b%** iff **a** < **b**. For the baseline case, *StressTool.exe* was just running idle applying no stress. Any CPU stress above 80% was considered unhealthy. *ProcessorTotal%ProcessorTime* and *ProcessorTotal%UserTime* counters were in **S1** with highest confidence (0.91 and 0.87 respectively), as expected. Most per-process counters for *agent.exe* and *StressTool.exe* also appeared in **S1** and **S2**. Many .NET counters were picked to our surprise. On closer inspection we realized that the stress tool used was written using managed C++ explaining the .NET counters.

Memory Stress: *Consume.exe* [W02] was run on the target machine to vary memory consumption 0 – 70%. Per-

process counters were monitored for *Consume.exe* and *agent.exe*. For baseline, *Consume.exe* remained idle. Any memory consumption above 60% was considered unhealthy. Memory counters for *Consume.exe* -- *VirtualBytesPeak*, *VirtualBytes*, *WorkingSetPeak*, *pageFileBytesPeak*, *PageFileBytes*, and *PrivateBytes*, all appeared at the top of **S1** (confidence of 1.0) and **S2** as expected.

Disk Stress: A disk stressor [W02] was used to write varying amounts of data to the disk. At unhealthy levels, the disk writes filled 99.95% of the total disk space. The counters in **S1** and **S2** were as expected with very few false positives and negatives.

I/O Stress: *sqlio.exe* [W02] was used to simulate different I/O patterns. The I/O patterns were specified using batch files and the rank-ordering of different stress levels were not obvious. Hence, tag-unordered data-categorization was used to index counter snapshots.

Network Stress: A disk stressor [W02] was adapted to write different amounts of data over the network with varying intervals. An idle network was used for baseline. To our surprise, PerfAnalyzer picked up *redirector* performance counters, which was not expected. But the network stress tool wrote over the network using a mapped network share and the Server Message Block (SMB) protocol, which is why the re-director performance counters were picked up.

SQL Server 2005: *StressSQL.exe* [W02] was used to stress SQL Server 2005. Per-process counters for various SQL Server processes such as *Sqlservr.exe* and *msmdsrv.exe* were monitored. A dummy database was created and a dummy set of queries were created to query the database. *StressSQL.exe* allows us to configure the number of users, user think-time, number of queries/user, etc. Various experiments were performed and rank-ordering was used

for data-categorization. The counters picked by PerfAnalyzer were a very good match for the expected SQL Server counters (expert specified [W01]).

IIS Web Server 7.0: Web application stress test tool, *was.exe* [W02], was used to stress an IIS 7.0 installation. Outside requests to a *.aspx* file were made using *was.exe*. Number of users, random delays, requests/user, etc. can be specified to the stress tool and different experiments were conducted at baseline, healthy, and unhealthy levels. We expected ASP.NET counters to be picked, but no counters were picked to our surprise. On closer inspection, we realized that the IIS Server served requests out of its cache for simple ASP pages. Many such interesting observations can be made by observing performance counters in **S1** and **S2**.

k	>10,000	f_{entropy}	0.1
 S 	~1000	f_{IGSU}	0.2
Δ	10s	2nd pass	OLR/MLR regression
m	50	$f_{\text{regression}}$	0.5
1st pass	Entropy + IGSU	$ V / D $	0.2

Figure 8: Experimental settings

Our quantitative evaluation results are displayed in Figure 9. We evaluated many combinations of first-pass and second-pass filters. For each combination, we evaluate various choices of data-categorization. However, in the interest of space, we only display the results for settings displayed in Figure 8. All data-sets except I/O were rank-ordered. I/O data-set was tag-unordered. For each subject, we started with $|S| \sim 1000$ counters. On an average over seven subjects, the size of **S2** reduced by about 91% compared to size of **S** (after second pass). There was a 78% size reduction observed between **S** and **S1** (after first pass), on an average. For validation, we randomly selected about 20% ($|V|/|D|$) of **S2** snapshots in **D** as test data-set. We had a prediction accuracy of over 98% for all the applications as shown in Figure 9.

Datasets	Pass 1 Filter # Counters	Pass 2 Filter # Counters	Prediction Accuracy
CPU Stress	223	113	100%
Memory Stress	204	76	100%
Disk Stress	273	136	98.93%
I/O Stress (sqlio)	156	52	100%
Network Stress	123	44	100%
SQL Server 2005	271	133	100%
IIS Web Server 7.0	274	107	99.43

Figure 9: Validation results. $|S| \sim 1000$

V. RELATED WORK

Various performance testing guidelines [W01], based on performance counters, exist for applications such as web servers and database servers. Our approach attempts to mine these expert guidelines from performance counter data-set. Our work is most similar to Cohen et al.'s approaches [CZG05] and [CGK05]. The first approach [CZG05] analyzes system-level metrics to deduce the system state and determine if the state is similar to a previous faulty state. The second approach [CGK05] analyzes instrumentation data from network services in order to forecast and diagnose failure conditions. Apart from performance slowdowns, security and robustness violations are also threats to reliable and dependable computing. Various *mining* approaches exist to uncover security and robustness bugs from software. These approaches either mine application source code [RGJ07] or run-time [YEBBD06] information such as execution traces. These approaches employ sequence mining techniques [AS95, GZ05] to infer bugs. PerfAnalyzer monitors application performance for slowdowns, which are also a major threat to reliable and dependable services. PerfAnalyzer is also a mining approach, which mines health model by monitoring performance counters during application run-time. PerfAnalyzer employs various correlation and regression algorithms [CCWA03, H98, S05, W03] to mine the health model from the counter data-set. Many approaches exist for software failure diagnosis [TLHXZ07] such as offline diagnosis [ZGZ03], dynamic software bug detection [HJ92], check-pointing/re-execution [OSSN02], and distributed systems fault localization. [HJ93] outlines a methodology to collect performance data for performance bottleneck analysis in large-scale parallel systems. In contrast, PerfAnalyzer adopts a generic view of all kinds of failures, including software failures, by analyzing performance counters.

VI. CONCLUSIONS

We observed that the first pass gets rid of ~80% of redundant or irrelevant counters without throwing away good predictors. PerfAnalyzer has a negligible false negative percentage. However, even after evaluating various first pass algorithms and heuristics, **S1** counters after first-pass had false positives – some unrelated counters were present in **S1**. Second-pass algorithms are very sensitive to false positives in **S1**. The health model improved when obvious false positives were manually removed from **S1**. Manually removing counters was possible to some extent for the five simple scenarios we

considered. We intend to invest our future efforts in exploring better first-pass heuristics to reduce human intervention. Also, extensive experiments are required to determine optimum values for m , ω , Δ , k , and f values for various first and second-pass filters. Counter snapshots were collected from a single target machine. To generalize this approach to multiple machines, the monitoring framework has to be extended to incorporate *roles* and *aggregation*. Roles identify the role of a machine in a huge infrastructure. For example, in a data-center, a set of machines will be dedicated for indexing only. It makes more sense to report related counters among different machines with same roles in an aggregated (for example, average value) fashion. Finally, it will be interesting to explore event-based health models (events refer to application run-time events), as a complementary approach to counter-based health models explored in this paper.

ACKNOWLEDGMENT

We thank Hunter Hudson and the members of his Engineering Services team at Microsoft Center for Software Excellence for providing us with the resources needed to implement PerfAnalyzer and conduct experiments. The monitoring agent was developed by the Engineering Services team. We thank Christopher Marsh, Matt Pietrek, Craig Schertz, Jack Shu, and Dick Wilbur for various discussions and help with the monitoring agent and related experiments. The stress tools employed in our experiments were developed by several teams [W02] within Microsoft and we acknowledge their use here. Finally, we thank SQL Server Data Mining team members at Microsoft for valuable inputs and discussions on using business intelligence and data mining capabilities of SQL Server.

REFERENCES

- [A02] A. Agresti, "Categorical data analysis", Wiley Interscience, 2002
- [AS95] R. Agrawal, R. Srikant, "Mining sequential patterns", ICDE 1995
- [B07] C. Bishop, "Pattern recognition and machine learning", Springer, 2007
- [CCWA03] J. Cohen, P. Cohen, S. West, L. Aiken, "Applied multiple regression/correlation analysis for the behavioral sciences", LEA Inc., 2003
- [CGK05] I. Cohen, M. Goldszmidt, T. Kelly, J. Symons, J. Chase, "Correlating instrumentation data to system states: A building block for automated diagnosis and control", OSDI 2004
- [CKFFB02] M. Chen, E. Kiciman, E. Fratkin, A. Fox, E. Brewer, "Pinpoint: Problem determination in large dynamic systems", DSN 2002
- [CZG05] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, A. Fox, "Capturing, indexing, clustering, and retrieving system history", SOSP 2005
- [GZ05] G. Grahne, J. Zhu, "Fast algorithms for frequent itemset mining using FP-trees", IEEE TKDE 2005
- [H98] M. Hall, "Correlation-based feature selection for discrete and numeric class machine learning", ICML 2000
- [HJ92] R. Hastings, B. Joyce, "Purify: Fast detection of memory leaks and errors", USENIX 1992
- [HM93] J. Hollingsworth, B. Miller, Dynamic control of performance monitoring on large scale parallel systems, SIGARCH SC 1993
- [OSSN02] S. Osman, D. Subhraveti, G. Su, J. Nieh, "The design and implementation of Zap: A system for migrating computing environments", OSDI 2002
- [RGJ07] M. Ramanathan, A. Grama, S. Jagannathan, "Static specification inference using predicate mining", PLDI 2007
- [TLHXZ07] J. Tucek, S. Lu, C. Huang, S. Xanthos, Y. Zhou. "Triage: Diagnosing production run failures at the user's site", SOSP 2007
- [W01] MSDN Library, <http://msdn.microsoft.com>
- [W02] Microsoft Toolbox, Microsoft Intranet Resource
- [YEBBD06] J. Yang, D. Evans, D. Bharadwaj, T. Bhat, M. Das, "Perracotta: Mining temporal API rules from imperfect traces", ICSE 2006
- [ZGZ03] X. Zhang, R. Gupta, Y. Zhang, "Precise dynamic slicing algorithms", ICSE 2003