Predicting Resource Usage of Arbitrary Network Traffic Queries

Pere Barlet-Ros[⋆], *Gianluca Iannaccone*[†], *Josep Sanjuàs-Cuxart*[⋆],
Diego Amores-López[⋆], *Josep Solé-Pareta*[⋆]

[⋆] *Universitat Politècnica de Catalunya*
Barcelona, Spain

[†] *Intel Research*
Cambridge, UK

Abstract

Monitoring and mining real-time network data streams is crucial for managing and operating data networks. The information that network operators desire to extract from the network traffic is of different size, granularity and accuracy depending on the measurement task (e.g., relevant data for capacity planning and intrusion detection are very different). To satisfy these different demands, a new class of monitoring systems is emerging to handle multiple arbitrary and continuous traffic queries. Such systems must cope with the effects of overload situations due to the large volumes, high data rates and bursty nature of the network traffic – the alternative of provisioning them to handle peak rates is prohibitively expensive.

In this paper, we present the design and evaluation of a system that can accurately predict the resource usage needs of network traffic queries, even in the presence of extreme and highly variable traffic conditions. The novelty of our system is that it is able to operate without any explicit knowledge of the traffic queries. Instead, it extracts a set of features from the traffic streams to build a prediction model of the query resource requirements with deterministic (and small) worst case computational cost.

We present experimental evidence of the performance and robustness of our prediction methodology using real traffic traces and injecting synthetic traffic anomalies. Our results show that the system predicts the resources required to run each traffic query with small errors in all the various traffic scenarios.

This prediction can later be used for load shedding purposes in order to allow current network monitoring systems to quicky react to overload situations by sampling the incoming packet streams or providing a summarized view of the input traffic.

1 Introduction

Network monitoring applications that must extract a large number of real-time metrics from many input streams are becoming increasingly common. These include for example applications that correlate network data from multiple sources (e.g., end-systems, access points, switches) to identify anomalous behaviors, aid in traffic engineering, capacity planning or managing and troubleshooting the network.

The main challenge in these systems is to keep up with ever increasing input data rates and processing requirements. Data rates are driven by the increase in network link speeds, application demands and the number of end-hosts in the network. The processing requirements are growing to satisfy the demands for fine grained and continuous analysis, tracking and inspection of network traffic. This challenge is made even harder as network operators expect the queries to return accurate enough results in the presence of anomalous traffic patterns when the system is under additional stress (and the query results are most valuable!).

Recently, several research proposals have addressed this challenge [17, 20, 21, 6, 12]. The solutions introduced belong to two broad categories. The first includes solutions that consider a pre-defined set of metrics and can report approximate results (within given accuracy bounds) in the case of overload [17, 12]. The second category includes solutions that define a declarative query language with a small set of operators for which the resource usage is assumed to be known [20, 21, 6]. In the presence of overload, operator-specific load shedding techniques are implemented (e.g., selectively discarding some records, computing approximate summaries) so that the accuracy of the entire query is preserved within certain bounds.

These solutions present two common limitations: (i) they restrict the types of metrics that can be extracted from the traffic streams limiting therefore the possible applications and uses of those systems, and (ii) they assume explicit knowledge of the cost and selectivity of each operator, requiring a very careful and time-consuming design and implementation phase for each of them.

In this paper, we present a system that supports multiple arbitrary and continuous traffic queries on the input streams. The system can predict overload situations due to anomalous, extreme or highly variable traffic mixes. The core of our methodology consists in the real-time modeling and prediction of the system resource usage, which allows the monitoring system to *anticipate* future bursts in the resource requirements. The novelty of our approach is that it does not require any explicit knowledge of the query or of the types of computations it performs (e.g., flow classification, maintaining aggregate counters, string search). Thus, any load shedding scheme based on our prediction methodology would preserve the flexibility of the monitoring system and would allow fast implementation and deployment of new network data mining applications.

Since we have no knowledge of the computations performed on the packet streams, we need to infer those from the relation between a large set of pre-defined "features" of the input stream and the actual resource usage. A feature is a counter that describes a specific property of a sequence of packets (e.g., number of unique source IP addresses). The features we compute on the input stream have the advantage of being lightweight with a deterministic worst case computational cost. Then, we automatically identify those features that best model the resource usage of each query and use them to predict the overall load of the system. If the prediction exceeds a threshold, we could then implement several load shedding techniques, such as packet sampling, flow sampling or computing summaries of the data streams to reduce the amount of resources required by the query to run.

For simplicity, in this paper we focus only on one resource: the CPU cycles. However, we believe that our approach can be also applied to other system resources as well (e.g., memory, disk space, disk bandwidth).

We have integrated our prediction mechanism into the CoMo monitoring system [14] and we evaluated it using packet traces collected on an research ISP network and running a set of seven concurrent queries on the packet streams that range from maintaining simple counters (e.g., number of packets, application breakdown) to more complex data structures (e.g., per-flow classification, ranking of most popular destinations or string search). In addition, we introduced several anomalies into the packet traces to emulate different network attacks to other systems in the network as well as targeted against the monitoring system itself.

The remainder of this paper is structured as follows. The next section presents in greater detail some related work. Section 3 introduces the monitoring system we use for our study and the set of queries that will be used throughout the paper. We describe our prediction method in detail in Section 4 and validate its performance using real traffic traces in Section 5. Section 6 compares our approach to other approaches based on time series analysis and presents experimental results showing how the system can handle anomalous traffic patterns and the overhead introduced by the prediction. Finally, Section 7 concludes the paper and introduces several ideas for future work.

2 Related Work

The design of mechanisms to handle overload situations is a classical problem in any real-time system design and several previous works have proposed solutions to the problem.

In the network monitoring space, NetFlow [7] is considered the state-of-the-art. It is a widely deployed, general purpose solution supported in most of today's routers. It extracts pre-defined per-flow information (depending on the version of Netflow) and periodically reports to a central collection server. In order to handle the large volumes of data exported and to reduce the load on the router it resorts to packet sampling. The sampling rate must be defined at configuration time, and to handle unexpected traffic scenarios network operators tend to set it to a low "safe" value (e.g., 1/100 or 1/1000 packets). Adaptive NetFlow [12] allows routers to dynamically tune the sampling rate to the memory consumption in order to maximize the accuracy given a specific incoming traffic mix. Keys et al. [17] extend the approach used in NetFlow by extracting and exporting a set of 12 traffic summaries that allow the system to answer a fixed number of common questions asked by network operators. They deal with extreme traffic conditions using adaptive sampling and memory-efficient counting algorithms. Our work differs from this approach in that we are not limited to a small set of known traffic summaries but instead we can handle arbitrary network data mining applications. Our method does not require any explicit knowledge of what information the application is trying to extract from the traffic streams.

Several research proposals in the stream database literature are also very relevant to our work. The Aurora system [4] can process a large number of concurrent queries that are built out of a small set of operators. In Aurora, load shedding is achieved by inserting additional drop operators in the data flow of each query [21]. In order to find the proper location to insert the drop operators, [21] assumes explicit knowledge of the cost and selectivity of each operator in the data flow. Additional extensions have proposed mechanisms to have the drop operator discard the "right" records from the stream [9, 16]. In [5, 20], the authors propose a system that applies approximate query processing techniques, instead of dropping records, to provide approximate and delay-bounded answers in presence of overload.

Again, our work differs from these approaches in that we have no explicit knowledge on the query and therefore we cannot make any assumption on its cost or selectivity to know when it is the right time to drop records.

Our system is based on extracting features from the traffic streams with deterministic worst case time bounds. Several solutions have been proposed in the literature to this end. For example, counting the number of distinct items in a stream has been addressed in the past in [13, 1]. In this work we implemented the multi-resolution bitmap algorithms for counting flows proposed in [13].

Finally, the overall design issues involved in building network monitoring systems that allow declarative queries as well as arbitrary traffic queries are addressed in [8, 14].

3 System Overview

In this section we describe the goals and challenges involved in the design of a prediction mechanism for arbitrary network data mining applications. We also introduce the monitoring platform and the set of queries we use throughout the paper

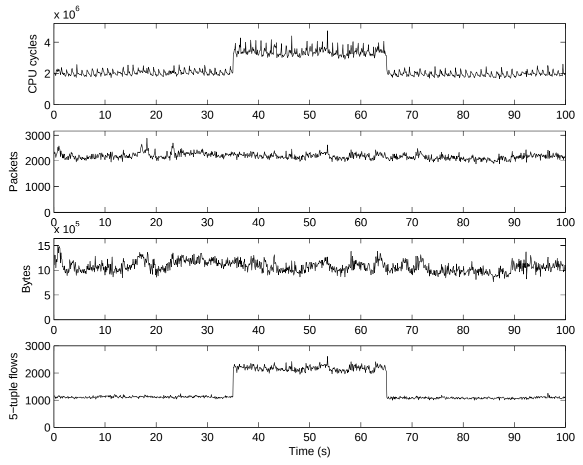

Figure 1: CPU usage compared to the number of packets, bytes and flows

for evaluating our method.

Our thesis is that the cost of maintaining the data structures needed to execute a query can be modeled by looking at a set of traffic features that characterizes the input data. The intuition behind this thesis is that each query incurs a different overhead when performing basic operations on the state it maintains while processing the input packet stream such as, for example, creating new entries, updating existing ones or looking for a valid match. We argue that the time spent by a query is often dominated by the overhead of some of these operations and therefore can be modeled by considering the right set of simple traffic features.

A traffic feature is a counter that describes a property of a sequence of packets. For example, potential features could be the number of packets or bytes in the sequence, the number of unique source IP addresses, etc. In this paper we will select a large set of simple features that have the same underlying property: deterministic worst case computational complexity. Later we will describe how a large set of features can be efficiently extracted from the traffic stream (Section 4.1).

Once we extract a large number of features from the traffic stream, the challenge is in identifying the right ones that can be used to accurately model and predict the query's CPU usage. Figure 1 illustrates a very simple example. The figure shows the time series of the CPU cycles consumed by an "unknown" query (top graph) when running over a $100s$ snapshot of our data set (described in Section 3.3), where we inserted an artificially generated anomaly. The three bottom plots show three possible features over time: the number of packets, bytes and flows (defined by the classical 5-tuple: source and destination addresses, source and destination port numbers and protocol number). It is clear from the figure that the bottom plot would give us the most useful information to predict the CPU usage over time for this query. It is also easy to infer that the query is performing some sort of per-flow classification, hence the higher cost when the number

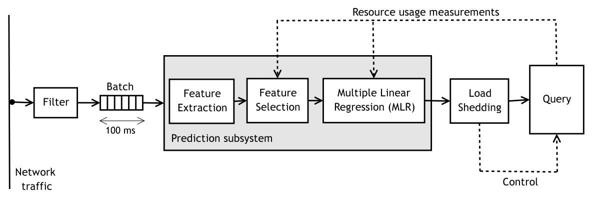

Figure 2: System overview

of flows increases despite the volume of packets and bytes remains fairly stable.

Based on this observation, we designed a method that automatically selects the most relevant feature(s) from a small sequence of packets and uses them to accurately predict the CPU usage of arbitrary queries. This fine-grained and short-term prediction could then be used to quickly react to overload situations by sampling the input streams or by providing a summarized view of the traffic data.

3.1 Monitoring platform

We chose the CoMo platform [14] for developing and evaluating our resource usage prediction method. The platform allows users to define traffic queries as plug-in modules written in C language. The user is required to specify a simple stateless filter to be applied on the incoming packet stream (it could be all the packets) and also the granularity of the measurements, hereafter called *measurement interval* (i.e., the time interval that will be used to report continuous query results). Then all complex stateful computations are contained within the plug-in module code.

In order to provide the user with the maximum flexibility when specifying queries, CoMo does not restrict the type of computations that a plug-in module can perform. As a consequence, the platform does not have any explicit knowledge of the data structures used by the plug-in modules or the cost of maintaining them. Therefore, our load shedding mechanism must operate only with external observations of the CPU, memory and bandwidth requirements of the modules – and these are not known in advance but only after a packet has been processed.

Figure 2 shows the components and the data flow in the system. The prediction subsystem (in gray) intercepts the packets from the filter before they are sent to the plug-in module implementing the traffic query. To implement the prediction we also instrumented the core platform to export some performance metrics. In this paper we focus only on the CPU cycles consumed by the query.

The system operates in four phases that are executed online. First, it groups each $100ms$ of traffic in a "batch" of packets[1]. Each batch is then processed to extract a large pre-

[1]The choice to use batches of 100ms is somewhat arbitrary. Our goal is not to delay excessively the query results but at the same time use a time interval large enough to observe a meaningful number of packets. Indeed an interval too small would add a significant amount of noise in the system

Name	**Description**	**Data structures**	**Dominant resource**
application	Port-based application classification	Array of counters	CPU
flows	Per-flow counters	Hash table	CPU, memory
high-watermark	High watermark of link utilization	Aggregated counters	CPU
link-count	Traffic load	Aggregated counters	CPU
popular destinations	Per-flow counters for the 10 most popular destination IPs	Hash table, sorted lists	CPU, memory
string search	Identifies a sequence of bytes in the payload	Linked list	CPU, disk bandwidth
trace	Full-payload collection	Linked list	Disk bandwidth

Table 1: Queries used in the experimental evaluation

Trace name	Start Time	Duration	File size	Packets	Bytes	Avg load	Max load	Min load
w/o payloads	Wed Nov 02, 2005 16:30	30 min.	8.29 GB	103.7 M	81.1 GB	360.46 Mbps	483.28 Mbps	197.32 Mbps
with payloads	Tue Apr 11, 2006 08:00	30 min.	30.87 GB	49.4 M	29.9 GB	133.04 Mbps	212.22 Mbps	96.15 Mbps
Abilene	Thu Aug 14, 2002 09:00	120 min.	34.08 GB	532.4 M	370.6 GB	411.90 Mbps	623.78 Mbps	286.22 Mbps
CENIC	Thu Mar 17, 2005 15:50	30 min.	3.82 GB	59.5 M	56.04 GB	249.62 Mbps	936.86 Mbps	79.05 Mbps

Table 2: Traces used in the experiments

defined set of traffic features (Section 4.1). The feature selection subsystem is in charge of selecting the most relevant features for prediction purposes according to the recent history of the query's CPU usage (Section 4.3). This subset of relevant features is then given as input to the multiple linear regression subsystem to predict the CPU cycles required by the query to process the entire batch (Section 4.2).

If the prediction exceeds the current allocation of cycles, a load shedding subsystem could pre-process the batch to discard (e.g., via packet or flow sampling) a portion of the packets. Finally, the actual CPU usage is computed and fed back to the prediction subsystem to close the loop (Section 4.4).

3.2 Queries

Despite the fact that the actual metric computed by the query is not relevant for our work – our system does not make any assumptions about it – we are interested in considering a wide range of queries when performing the evaluation. We have selected a set of queries that are part of the standard distribution of CoMo[2]. Table 1 provides a brief summary of the queries.

Three queries (*link-count*, *application* and *high-watermark*) maintain simple arrays of counters depending on the timestamps of the packets (and port numbers for *application*). The cost of running these queries is therefore driven by the number of packets. The *trace* query stores the full payload of all packets that match a stateless filter rule and therefore the cost depends on the number of bytes to be stored. The query *string search* stores all packets that contain a given string. It uses the Boyer-Moore algorithm [3] where the cost is linear with the number of bytes processed. Finally *flows* and *popular destinations* perform a flow classification and maintain a large number of per-flow counters for all the flows (in a way similar to NetFlow) or just the ones that exchanged the largest number of bytes, respectively. The cost in this last two queries depend on the number of flows in the packet stream but also on the details of the data structures used for the classification.

We believe that the set of queries we have chosen form a representative set of typical uses of a real-time network monitoring system and present different CPU usage profiles for the same input traffic. In the next sections we will show that our approach is general enough to handle efficiently all these different cases in normal and extreme traffic scenarios.

3.3 Datasets

Our testbed equipment consists of a PC with an Endace DAG 4.3GE card [11] equipped with 2 network interfaces. Through a pair of optic splitters, the card receives a copy of the traffic of a full-duplex 1 Gbps link that connects the Catalan academic network (also known as the Scientific Ring) with its Spanish counterpart (RedIris).

For our testing purposes we collected two 30-minute traces from one of the link directions, in November 2005 and April 2006. In the first trace, we only collected the packet headers, while in the second one the full packet payloads were acquired.

In order to study our method in other environments, we extend our dataset with two anonymized packet header traces collected by the PMA project of NLANR [18], in August 2002 and March 2005. The first one (Abilene) consists of a OC48c Packet-over-SONET unidirectional trace collected at the Indianapolis router node of the Abilene backbone (eastbound towards Cleveland). The second trace (CENIC) consists of the first 30 minutes of the data set collected on the 10

and increase the prediction error. Our results indicate that 100ms represents a good trade-off between accuracy and delay. However, this is clearly a function of the input traffic traces we used. We leave the investigation on the proper batch duration for future work.

[2]The actual source code of each query is available at `http://como.intel-research.net`.

Gigabit CENIC HPR backbone link between Sunnyvale and Los Angeles. Details of the traces are presented in Table 2.

4 Prediction Methodology

In this section we describe in detail the three phases that our system executes to perform the prediction (i.e., *feature extraction*, *feature selection* and *multiple linear regression*) and how the resource usage is monitored. The only information we require from the continuous query is the measurement interval of the results. Avoiding the use of additional information increases the range of applications where this approach can be used and also reduces the likelihood of compromising the system by providing incorrect information about a query.

4.1 Feature Extraction

We are interested in finding a set of traffic features that are simple and inexpensive to compute, while helpful to characterize the CPU usage of a wide range of queries. A feature that is too specific may allow to predict a given query with great accuracy, but could even cost as much as directly answering the query (e.g., counting the packets that contain a given pattern in order to predict the cost of signature-based IDS-like queries). Our goal is therefore to find features that may not explain in detail the entire cost of a query, but can still provide enough information about the aspects that dominate the processing cost. For instance, in the previous example of a signature-based IDS query, the cost of matching a string will mainly depend on the number of collected bytes.

In addition to the number of packets and bytes, we maintain four counters per *traffic aggregate* that are updated every time a batch is received. A traffic aggregate considers one or more of the IP header fields: source and destination IP addresses, source and destination port numbers and protocol number. For example, we may aggregate packets based on the source IP address and source port number, where each aggregate (or "item") is made of all packets that share the same source IP address and source port number pair. The four counters we monitor per aggregate are: (i) the number of unique items in a batch; (ii) the number of new items compared to all items seen in a measurement interval; (iii) the number of repeated items in a batch (i.e., items in the batch minus unique) and (iv) the number of repeated items compared to all items in a measurement interval (i.e., items in the batch minus new).

Table 3 shows the combinations of the five header fields considered in this work. Although we do not evaluate other choices here, we note that other features may be useful (e.g., source IP prefixes or other combinations of the 5 header fields). Adding new traffic features (e.g., payload-related features) as well as considering other combinations of the existing ones constitutes an important part of our future work. However, we will address the trade-off between the number of features and the overhead of running the prediction in greater detail in Section 5.

1	src-ip
2	dst-ip
3	protocol
4	<src-ip, dst-ip>
5	<src-port, proto>
6	<dst-port, proto>
7	<src-ip, src-port, proto>
8	<dst-ip, dst-port, proto>
9	<src-port, dst-port, proto>
10	<src-ip, dst-ip, src-port, dst-port, proto>

Table 3: Traffic aggregates

This large set of features (four counters per traffic aggregate plus the total packet and byte counts, i.e., 42 in our experiments) helps narrowing down which basic operations performed by the queries dominate their processing costs (e.g., creating new entries, updating existing ones or looking up for entries). For example, the new items are relevant to predict the CPU requirements of those queries that spend most time creating entries in the data structures. The repeated items feature may be most relevant to queries where the cost of updating the data structures is much higher than the cost of creating them.

In order to extract the features with the minimum overhead, we use the multi-resolution bitmap algorithms proposed in [13]. The advantage of the multi-resolution bitmaps is that they bound the number of memory accesses per packet when compared to classical hash tables and they can handle a large number of items with good accuracy and smaller memory footprint than linear counting [22] or bloom filters [2].

We use two bitmaps for each aggregation level: one that keeps the per-batch unique count and the other that maintains the new count per measurement interval. The bitmap used to estimate the unique items must be updated per packet. Instead, the one used to estimate the new items can be simply updated per batch by just doing a bitwise OR with the bitmap used to mantain the unique count, because exactly the same bits will be set in the two bitmaps. The only difference between them is when they are reset to 0. As mentioned earlier, it is straightforward to derive the number of repeated and batch-repeated items from the counts of new and unique items respectively keeping just two additional counters. We dimension the multi-resolution bitmaps to obtain counting errors around 1% given the link speeds in our testbed.

4.2 Multiple Linear Regression

Regression analysis is a widely applied technique to study the relationship between a response variable Y and one or more predictor variables $X_1, X_2, \ldots, X_p$. The linear regression model assumes that the response variable Y is a linear function of the p X_i predictor variables[3]. The fact that this

[3]It is possible that the CPU usage of other queries may exhibit a non-linear relationship with the traffic features. A solution in that case may

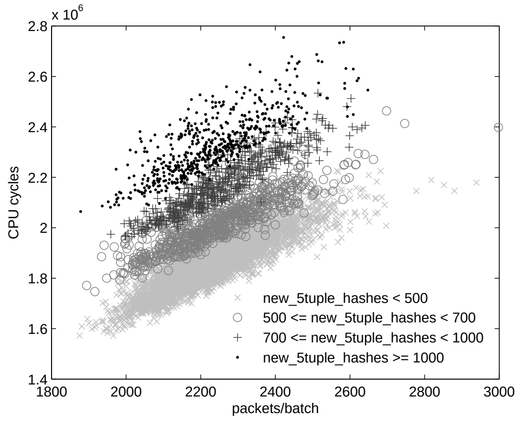

Figure 3: Scatter plot of the CPU usage and the number of packets in the batch (*flows* query)

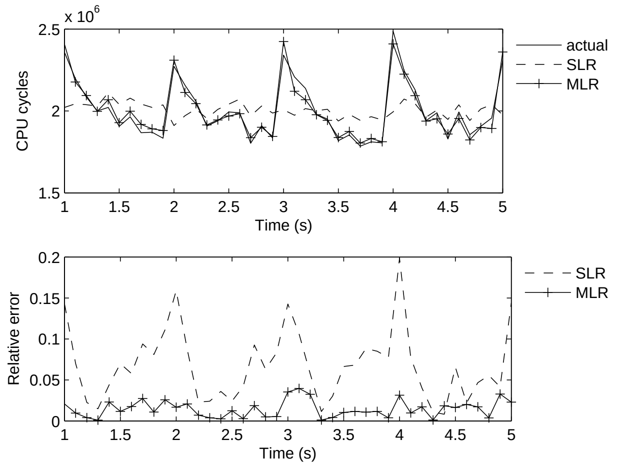

Figure 4: SLR versus MLR prediction (*flows* query)

relationship exists implies that any knowledge we have about the predictor variables provides us information about the response variable. Thus, this knowledge can be exploited for predicting the expected value of Y when the values of the p predictor variables are known. In our case, the response variable is the CPU usage while the predictor variables are the individual features.

When only one predictor variable is used, the regression model is often referred to as simple linear regression (SLR). Using just one predictor has two major drawbacks. First, there is no single predictor that gives good performance for all queries. For example, the CPU usage of the *link counts* query is well modeled by looking at the number of packets in each batch while the *trace* query is better modeled by the number of bytes in the batch. Second, the CPU usage of more complex queries may depend on more than a single feature.

To illustrate this latter point, we plot in Figure 3 the CPU usage for the *flows* query versus the number of packets in the batch. As we can see, there are several underlying trends that depend both on the number of packets and on the number of new 5-tuples in the batch that SLR cannot consider. This is due to the particular implementation of the *flows* query that maintains a hash table to keep track of the flows and expires all flows at the end of each measurement interval.

Figure 4 shows that, for the *flows* query, the prediction error by using SLR is relatively large. The spikes in the CPU usage at the beginning of each measurement interval ($1s$ in this example) are due to the fact that when the table is empty, the number of new entries to be created is much larger than usual. This error could be much more significant in presence of traffic anomalies that abruptly increase the number of new entries to be created.

be to define new features computed as non-linear combinations of simple features. Studying specific network data mining applications that exhibit non-linear relationship with the set of features we have identified so far constitutes an important part of our future work.

Multiple linear regression (MLR) extends the simple linear regression model to several predictor variables. MLR is used to extract a linear combination of the predictor variables that is maximally correlated with the response variable. The general form of a linear regression model for p predictor variables can be written as follows [10]:

$$\begin{aligned} Y_i = \beta_0 + \beta_1 X_{1i} + \beta_2 X_{2i} + \dots \\ \dots + \beta_p X_{pi} + \varepsilon_i, \qquad i = 1, 2, \dots, n \end{aligned} \tag{1}$$

where β_0 denotes the *intercept*, $\beta_1, \dots, \beta_p$ are the *regression coefficients* that need to be estimated and ε_i is the *residual term* associated with the i-th observation. The residual term is an unobservable random variable that represents the omitted variables that affect the response variable, but that are not included in the model.

In fact, Equation 1 corresponds to a system of equations that in matrix notation can be written as [10]:

$$\begin{pmatrix} Y_1 \\ Y_2 \\ \vdots \\ Y_n \end{pmatrix} = \begin{pmatrix} 1 & X_{11} & \dots & X_{p1} \\ 1 & X_{12} & \dots & X_{p2} \\ \vdots & \vdots & & \vdots \\ 1 & X_{1n} & \dots & X_{pn} \end{pmatrix} \begin{pmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_p \end{pmatrix} + \begin{pmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \vdots \\ \varepsilon_n \end{pmatrix}$$

or simply

$$Y = X\beta + \varepsilon \tag{2}$$

where

- Y is a $n \times 1$ column vector of the response variable observations. We obtain the values of Y by measuring the CPU usage of the previous n batches processed by the query;

- X is a $n \times (p + 1)$ matrix resulting from n observations of the p predictor variables $X_1, \dots, X_p$ (the first column of 1's represents the intercept term β_0). That is, the values of the p features we extracted from the previous n batches;

- β is a $(p+1) \times 1$ column vector of unknown parameters $\beta_0, \beta_1, \ldots, \beta_p$, where $\beta_1, \ldots, \beta_p$ are referred to as the regression coefficients or weights;

- and ε is a $n \times 1$ column vector of n residuals ε_i.

The regression model presented in the Equation 1 assumes an infinite number of observations has been collected. In practice, the number of observations is very small and the regression coefficients β and the residuals ε have to be estimated. The estimators b of the regression coefficients β are obtained by the Ordinary Least Squares (OLS) procedure, which consists of choosing the values of the unknown parameters $b_0, \ldots, b_p$ in such a way that the sums of squares of the residuals is minimized. In our implementation, we use the singular value decomposition (SVD) method [19] to compute the OLS. Although SVD is more expensive than other methods, it is able to obtain the best approximation, in the least-squares sense, in the case of an over- or underdetermined system.

The statistical properties of the OLS estimators lie on some assumptions that must be fulfilled [10, pp. 216]: (i) the rank of X is $p+1$ and is less than n, i.e., there are no exact linear relationships among the X variables (no multicollinearity); (ii) the variable ε_i is normally distributed and the expected value of the vector ε is zero; (iii) there is no correlation between the residuals and they exhibit constant variance; (iv) the covariance between the predictors and the residuals is zero. In Section 4.3 we present a technique that makes sure the first assumption is valid. We have also verified experimentally on the packet traces that the other assumptions hold but in the interest of space we will not show the results here.

Going back to the example of the *flows* query, Figure 4 shows the prediction accuracy when using MLR with the number of packets and new 5-tuples as predictors.

However, since we assume arbitrary queries, we do not know which features have to be used as predictors in the MLR. It would be possible to use all the extracted traffic features in the regression, since MLR should be able to find a combination of them that is maximally correlated with the CPU usage. However, as it can be deduced from Equation 2, the cost of MLR does not depend only on the amount of history used to compute the linear regression (i.e. n), but also on the number of variables used as predictors (i.e. p). If a large number of predictors is used, the cost of the MLR would increase significantly and it could impose too much overhead to the prediction process. Next section presents a technique that is used to select only the subset of traffic features that are more relevant to a given query.

4.3 Feature Selection

Including in the regression model as many predictor variables as possible has several drawbacks [10]. As we said before, the cost of the linear regression increases with the number of predictors included in the model and the gain the additional predictors bring does not justify their cost. In addition, even if we were able to include in the model all the possible predictors, there would still be a certain amount of randomness that cannot be explained by any predictor. Finally, by introducing redundant predictors into the model (i.e. predictors that are linear functions of some other predictors) the non-multicollinearity assumption would not be met. It is worth to note that some predictors may become redundant under some traffic conditions (e.g., the number of connections and packets under a SYN-flood attack).

Once a choice of the features to compute on the batch is made, it is important to identify a small subset of features to be used as predictors. In order to support arbitrary queries, we need to define a generic feature selection algorithm. Most of the algorithms proposed in the literature are based on a sequential variable selection procedure [10]. However, they are usually too expensive to be used in a real-time system. For this reason we decided to use a variant of the Fast Correlation-Based Filter (FCBF) [23], which can effectively remove both irrelevant and redundant features and is very computationally efficient.

Our variant differs from the original FCBF algorithm in that we use the *linear correlation coefficient* (Equation 3) as a predictor goodness measure, instead of the *symmetrical uncertainty* measure [23], which is based on the information-theoretical concept of *entropy*. The algorithm consists of two main phases:

1. *Selecting relevant predictors:* The linear correlation coefficient between each predictor and the response variable is computed as follows:

$$r = \frac{\sum_i (X_i - \overline{X_i})(Y_i - \overline{Y_i})}{\sqrt{\sum_i (X_i - \overline{X_i})^2}\sqrt{\sum_i (Y_i - \overline{Y_i})^2}} \qquad (3)$$

   The predictors with a coefficient below a predefined *FCBF threshold* are discarded as not relevant. In Section 5.1 we will address the problem of choosing the appropriate *FCBF threshold*. This phase has a time complexity that grows linearly with the number of predictors.

2. *Removing redundant predictors:* The predictors that are left after the first phase are ranked according to their coefficient values and processed iteratively to discard predictors that have a mutual strong correlation. Each iteration starts from the first element of the list (i.e. the predictor with the highest linear correlation coefficient) and computes the correlation coefficients between this element and all the remaining predictors. For each pair of predictors, if their relative correlation coefficient is higher than the correlation between the predictors and the response variable (computed in the previous phase), the predictor lower in the list is removed as redundant. Then, the algorithm continues starting again from the

second predictor. In each iteration the algorithm can usually remove a significant number of redundant features, giving this phase a time complexity of $O(logN)$, where N is the number of predictors in the list.

4.4 Measurement of System Resources

Measuring CPU usage is not an easy task. The mechanisms provided by the operating system do not offer a good enough resolution for our purposes, while processor performance profiling tools [15] impose a large overhead and are not a viable permanent solution to fine grained CPU measurement.

In this work, we use instead the *time-stamp counter* (TSC) [15] to measure the CPU usage. Every batch of packets is sent to each query one-by-one. We read the TSC before and after executing each query, and the difference between them corresponds to the number of cycles used by the query to process the batch.

The CPU usage measurements that are fed back to the prediction system should be accurate and free of external noise to reduce the errors in the prediction. However, we detected that measuring CPU usage at very small timescales incurs in several sources of noise:

1. *Instruction reordering:* The processor can reorder instructions at run time in order to improve performance. In practice, the `rdtsc` instruction used to read the TSC counter is often reordered, since it simply consists of reading a register and it does not have dependencies with other instructions. In practice, the `rdtsc` instruction at the beginning of the query can be reordered with other instructions that do not belong to the query, while the one at the end of the query can be executed before the query actually ends. Both these events happen quite frequently and lead to inaccuracies in the measurements, so we execute a serializing instruction (e.g. `cpuid`) before and after our measurements, to avoid the effects of instruction reordering [15]. Since the use of serializing instructions can have a severe impact on the system performance, we only take two TSC readings per query and batch, and do not take any partial measurements during the execution of the query.

2. *Context switches:* The operating system may decide to schedule out the query process in the middle of two readings of the TSC. In that case, we would be measuring not only cycles belonging to the query, but also cycles of the process (or processes) that are preempting the query.

   In order to discard these measurements, we monitor two fields of the *rusage* process structure in the Linux kernel, called *ru_nvcsw* and *ru_nivcsw*, that count the number of voluntary and involuntary context switches respectively. If a context switch happens during a measurement, we discard that observation from the history and replace it with our prediction.

3. *Disk accesses.* Disk accesses can interfere with the CPU cycles needed to process a query. In CoMo, a separate process is responsible for scheduling disk accesses to read and write query results. In practice, since disk transfers are done asynchronously using DMA, memory accesses will compete for the system bus with the disk transfers. In Section 6 we show how disk accesses have a limited impact on the performance of the prediction system.

We do not take any particular action in the case of other causes of measurement noise, such as CPU frequency scaling or cycles executed in system mode, since we experimentally checked that they usually have much less impact on the CPU usage patterns than the sources of error described above. It is also important to note that all the sources of error we detected so far are independent from the input traffic. Therefore, they cannot be directly exploited by a malicious user trying to introduce errors in our CPU measurements to attack our monitoring system.

5 Validation

We show the performance of an actual implementation of our prediction method running on the packet traces (Table 2), as well as its sensitivity to the configuration parameters. In order to understand the impact of each parameter, we study the prediction subsystem in isolation from the sources of measurement noise identified in Section 4.4. We disabled the disk accesses in the CoMo process responsible for storage operations to avoid competition for the system bus. In Section 6, we evaluate our method in a fully operational system.

To measure the performance of our method we consider the relative error in the CPU usage prediction while executing the seven queries defined in Table 1 over the traces in our dataset. The relative error is defined as the absolute value of one minus the ratio of the prediction and the actual number of CPU cycles spent by the queries over each batch.

5.1 Prediction Parameters

In our system, two configuration parameters have an impact on the cost and accuracy of the predictions: the number of observations (i.e., n or the "history" of the system) and the FCBF threshold used to select the relevant traffic features. Here we analyze the more appropiate values of these parameters for the trace with full packet payloads, but almost identical values were obtained for the other traces in our dataset.

5.1.1 Number of observations

The cost (in terms of CPU cycles) of the linear regression directly depends on the history used, since every additional observation translates to a new equation in the system in (2). The accuracy of the prediction is also affected by the number of observations.

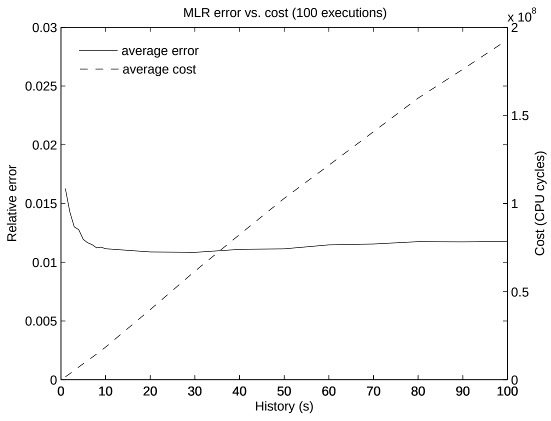

Figure 5: Prediction error versus cost as a function of the amount of history used to compute the prediction

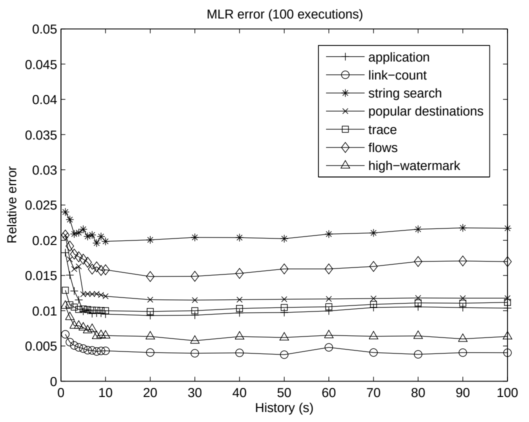

Figure 6: Prediction error as a function of the amount of history used to compute the prediction

In order to decide the appropriate amount of history to keep in our model, we ran multiple executions in our testbed with values of history ranging from $1s$ to $100s$ (i.e. from 10 to 1000 batches). We checked that histories older than $100s$ do not provide us any new relevant information for prediction purposes. Figure 5 shows the cost of computing the MLR and the prediction accuracy as a function of the amount of history (each observation corresponds to $100ms$ of traffic), while Figure 6 presents the prediction accuracy broken down by query.

As we can see the cost of computation grows linearly with the amount of history while the relative error between the prediction and the actual number of CPU cycles spent by the query stabilizes around 1.2% after $6s$ (i.e., 60 observations). Larger errors for very small amounts of history (e.g. $1s$) are due to the fact that the number of predictors (i.e. $p = 42$) is larger than the amount of history (i.e. $n = 10$ batches, 1 s) and thus the non-multicollinearity assumption is not met. Increasing the number of observations does not improve the

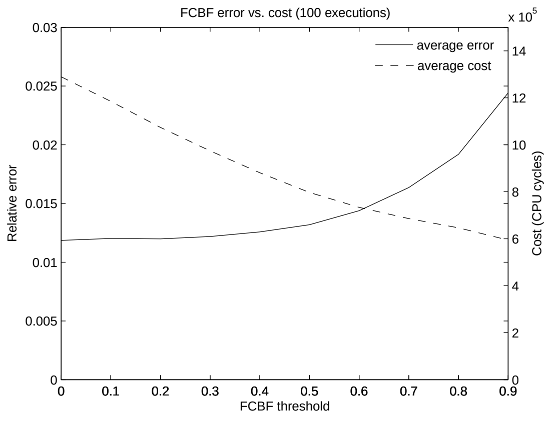

Figure 7: Prediction error versus cost as a function of the Fast Correlation-Based Filter threshold

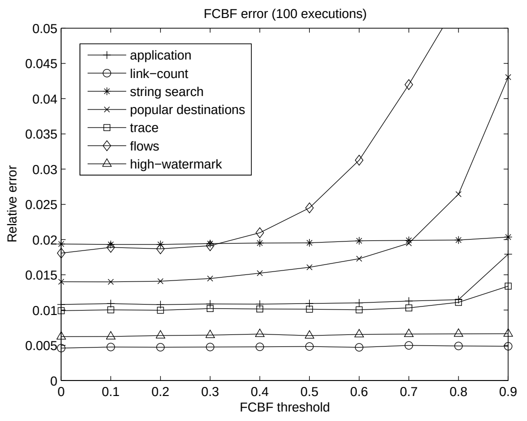

Figure 8: Prediction error as a function of the Fast Correlation-Based Filter threshold

accuracy because events that are not modeled by the traffic features are probably contributing to the error. However, a longer history makes the prediction model less responsive to sudden changes in the traffic that may change the behavior of a query as well as the most relevant features. In the rest of the paper, we use a number of observations equal to 60 (i.e., $6s$ of history).

5.1.2 FCBF threshold

This threshold determines which traffic features are relevant *and* not redundant in modeling the response variable. Large values of this threshold (i.e. closer to 1) will result on fewer features selected.

To understand the most appropriate value for the FCBF threshold, we ran multiple executions in our testbed with values of the threshold ranging from 0 (i.e., all features will be considered relevant but the redundant ones will still be removed) to 0.9 (i.e., most features are not selected). Figure 7 presents the prediction cost versus the prediction accuracy,

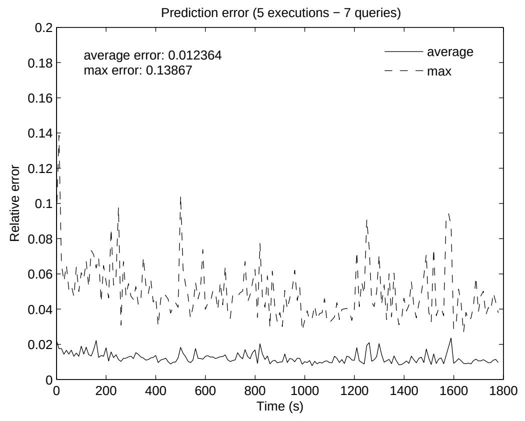

Figure 9: Prediction error over time (trace with payloads)

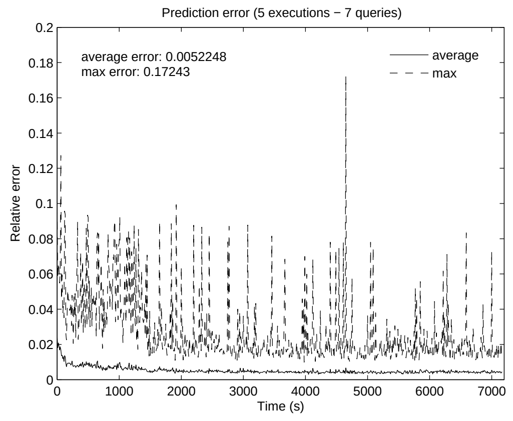

Figure 11: Prediction error over time (Abilene trace)

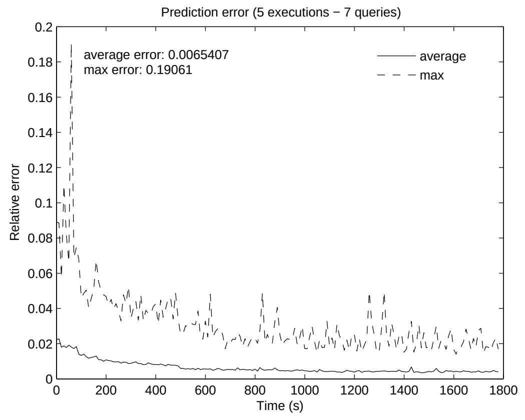

Figure 10: Prediction error over time (trace without payloads)

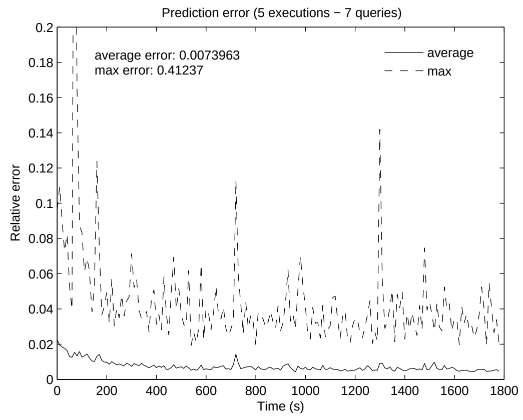

Figure 12: Prediction error over time (CENIC trace)

as a function of the threshold. The prediction cost includes both the cost of the selection algorithm and the MLR computation cost with the features selected by FCBF. Comparing this graph to Figure 5, we can see that using FCBF reduces the overall cost of the prediction by more than an order of magnitude (in terms of CPU cycles) while maintaining similar accuracy.

As the threshold increases, the value of p (see Section 4) decreases corresponding to a decrease in the CPU cycles needed to run the multiple linear regression. However the error remains fairly close to the minimum value (when all features are selected) and starts to ramp up only for relatively large values of the threshold (around 0.6). Very large values of the threshold (above 0.8) experience a much faster increase in the error compared to the decrease in the cost.

Lastly, in Figure 8 we plot the prediction accuracy broken down by query, as a function of the FCBF threshold. As expected, queries that can be well modeled with a single feature (e.g., *link-count*, *trace*) are quite insensitive to the particular value of the FCBF threshold, while queries that depend on more features (e.g., *flows*, *popular destinations*) exhibit a significant degradation in the accuracy of the prediction, when the FCBF threshold becomes closer to 0.9 (i.e., very few features are selected).

In the rest of the paper, we use a value of 0.6 for the FCBF threshold that achieves a good trade-off between prediction cost and accuracy for most queries.

5.2 Prediction Accuracy

In order to evaluate the performance of our method we ran the seven queries of Table 1 over the traces in our dataset. Figures 9 and 10 show the time series of the average and maximum error over five executions when running on the packet traces with and without payload, respectively.

The average error in both cases is consistently below 2%, while the maximum error reaches peaks of about 10%. These larger errors are due to external system events unrelated to the traffic that cause a spike in the CPU usage (e.g., cache misses) or due to a sudden change in the traffic patterns that are not appropriately modeled by the features that the prediction is using at that time. However, the time series show that our method is able to converge very quickly.

Trace with payloads

Query	Mean	Stdev	Selected features
application	0.0110	0.0095	packets, bytes
flows	0.0319	0.0302	*new* dst-ip, dst-port, proto
high-watermark	0.0064	0.0077	packets
link-count	0.0048	0.0066	packets
popular destinations	0.0169	0.0267	packets
string search	0.0198	0.0169	bytes
trace	0.0090	0.0137	bytes, packets

Trace without payloads

Query	Mean	Stdev	Selected features
application	0.0068	0.0060	*repeated* 5-tuple, packets
flows	0.0252	0.0203	*new* dst-ip, dst-port, proto
high-watermark	0.0059	0.0063	packets
link-count	0.0046	0.0053	packets
popular destinations	0.0136	0.0183	*new* 5-tuple, packets
string search	0.0098	0.0093	packets
trace	0.0092	0.0132	packets

Abilene trace

Query	Mean	Stdev	Selected features
application	0.0065	0.0068	packets
flows	0.0217	0.0174	*new* dst-ip, dst-port, proto
high-watermark	0.0046	0.0063	packets
link-count	0.0044	0.0063	packets
popular destinations	0.0154	0.0181	*new* src-dst-port, proto, pkts
string search	0.0116	0.0089	packets
trace	0.0090	0.0137	packets

Cenic trace

Query	Mean	Stdev	Selected features
application	0.0066	0.0083	packets
flows	0.0271	0.0341	packets, *new* 5-tuple
high-watermark	0.0058	0.0093	packets
link-count	0.0064	0.0110	packets
popular destinations	0.0218	0.0341	packets, *new* 5-tuple
string search	0.0272	0.0248	packets
trace	0.0079	0.0099	packets

Table 4: Breakdown of prediction error by query (5 executions)

The trace without payload (Figure 10) exhibits better performance, with average errors that drop well below 1%. This is well expected given that the trace contains only headers and the features we have selected allow to capture better the queries' CPU usage. Another interesting phenomenon is the downward trend in the maximum and average error. We conjecture that this is due to the kernel scheduler becoming more predictable and thus reducing the likelihood of external system events affecting our method.

Similar results are obtained for the two NLANR's traces as can be observed in Figures 11 and 12.

In Table 4, we show the breakdown of the errors by query. The average error is very low for each query with a relatively small standard deviation indicating compact distributions for the prediction errors. As expected, queries that make use of more complex data structures (e.g., *flows*, *popular destinations* and *string search*) incur in the larger errors but still at most around 3% on average.

It is also very interesting to look at the most frequent features that the selection algorithm identifies as most relevant for each query. Remember that the selection algorithm has no information about what computation the queries perform nor what type of packet traces they are processing.

The selected features give hints on what a query is actually doing and how it is implemented. For example, the number of bytes is the predominant traffic feature for the *string search* and *trace* queries when running on the trace with payloads. However, when processing the trace with just packet headers, the number of packets becomes the most relevant feature for these queries.

Another example worth noticing is the *popular destinations* query. In the trace with payloads it uses the number of packets as the most relevant predictor. This is an artifact of the particular location of the link where the trace was taken. Indeed, the trace is unidirectional and monitoring traffic destined towards the Catalan network. This results in a trace where the number of unique destination IP addresses is very small allowing the hash table used in the *popular destination* query to perform at its optimum with $O(1)$ lookup cost (hence the cost is driven by the number of packets, i.e. the number of lookups). This is not the case for the *flows* query that uses the destination port numbers as well thus increasing the number of entries (thus the lookup and insertion cost) in the hash table.

6 Evaluation

In this section we evaluate our prediction model in a fully operational system without taking any particular action in the presence of disk accesses. First, we compare the accuracy of our prediction model against two well-known prediction techniques, namely the Exponentially Weighted Moving Average (EWMA) and the Simple Linear Regression (SLR), in order to evaluate their performance under normal traffic conditions (Section 6.2). Then, we inject several synthetic anomalies in our traces in order to evaluate the robustness of the prediction techniques to extreme traffic conditions (Section 6.3). Finally, we discuss the cost of each component in our prediction subsystem, and present the overhead it imposes on the normal operations of the system (Section 6.4).

6.1 Alternative approaches

6.1.1 Exponentially Weighted Moving Average

EWMA is one of the most frequently applied time-series prediction techniques. It uses an exponentially decreasing weighted average of the past observations of a variable to predict its future values. EWMA can be written as:

$$\hat{Y}_{t+1} = \alpha Y_t + (1-\alpha)\hat{Y}_t \qquad (4)$$

where $\hat{Y}_{t+1}$ is the prediction for the instant $t+1$, which is computed as the weighted average between the real value of Y and its estimated value at the instant t, and α is the *weight*, also known as the *smoothing constant*.

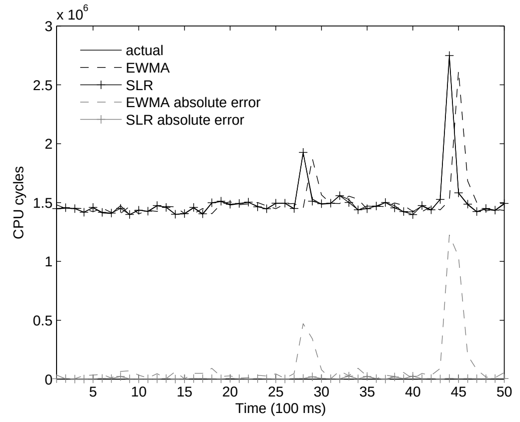

Figure 13: EWMA versus SLR prediction (*link-count* query)

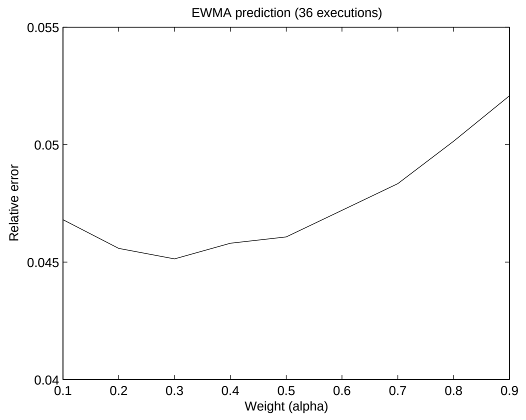

Figure 14: EWMA prediction error as a function of the weight (α)

In our case, we can use EWMA to predict the CPU requirements of a particular query at the instant $t + 1$, based on a weighted average of the cycles it used in the t previous observations. EWMA has the advantage of being easy to compute, but it only looks at the response variable (i.e. the CPU usage) to perform the prediction. A consequence of this is that EWMA cannot take into account variations in the input traffic to adjust its prediction accordingly. For example, if a batch contains a much larger number of bytes that the previous ones, EWMA will experience large errors for all queries that depend on the number of bytes (e.g., *string search*) and then slowly adapt to the traffic change.

Another example is presented in Figure 13. It shows that EWMA is not be able to anticiapte a significant increase in the CPU requirements of the *link-count* query when the number of packets suddenly increases, as can be observed at time 2.8 and 4.4 seconds. That is, EWMA predicts the effects (i.e. CPU usage) without taking into account the causes (i.e. the number of packets in the batch).

In order to pick the appropriate value of α we ran several experiments on the packets traces. The results presented in this section consider the best prediction accuracy we obtained that correspond to $\alpha = 0.3$, as shown in Figure 14.

6.1.2 Simple Linear Regression

SLR is a particular case of the multiple linear regression model, where one single prediction variable is used. The SLR model can be written as:

$$Y_i = \beta_0 + \beta_1 X_i + \varepsilon_i, \qquad i = 1, 2, \ldots, n \tag{5}$$

where X is the prediction variable, β_0 is the intercept, β_1 is the unknown coefficient and ε_i are the residuals. As in the case of MLR, the estimation b for the unknown β is obtained by minimizing the sum of the squared errors.

For those queries that depend on a single traffic feature, one would expect to obtain similar results to the ones obtained with our prediction model. However, without a feature selection algorithm, the best traffic feature to be used as predictor is not always known given the lack of explicit knowledge of the queries. In Table 4, we show that the most relevant traffic feature for most queries is the number of packets. Thus, in all the experiments we present in this section, we use the number of packets as predictor to compute the linear regression. The amount of history n is set to $6s$, as in the case of MLR.

In Figure 13, we can see that SLR can anticipate the increase in the CPU requirements of the *link-count* query, since its CPU usage only depends on the number of packets (Table 4).

6.2 Performance under Normal Traffic

Figure 15, 16 and 17 compare the prediction errors of the three methods running over the packet trace with payloads. Similar results were obtained for the other traces. The figures show the time series of the average error over 5 executions. The maximum error is computed over an interval of $10s$ and across the 5 executions. Table 5 shows the error for each individual query.

All three methods perform reasonably well with average errors around 10% for EWMA and SLR during the entire duration of the trace, and below 3% for MLR. This is expected given the stable traffic conditions that result in relatively stable CPU usage per query. The maximum errors are sometimes relatively large due to the frequent disk accesses – the trace query stores to disk all packets it receives. However, in Figure 18 we show the 95th-percentile of the prediction error, where we can see the limited impact of disk accesses on the prediction accuracy. Overall, the prediction error for MLR is smaller and more stable than for the other methods.

Inspecting Table 5 we can make two observations on the performance of EWMA and SLR. First, with EWMA the error is uniform across queries although the two queries that depend on the number of bytes (i.e. the packet sizes) experience higher variability in the prediction error. This confirms

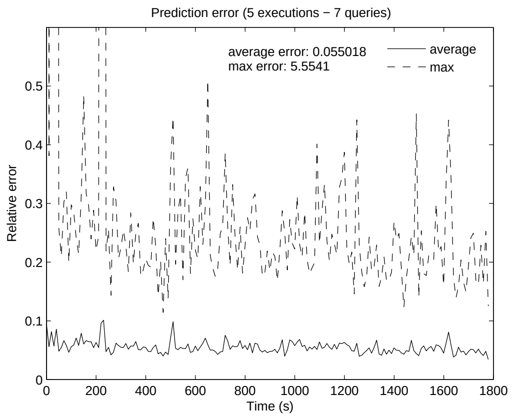

Figure 15: EWMA prediction error over time (trace with payloads)

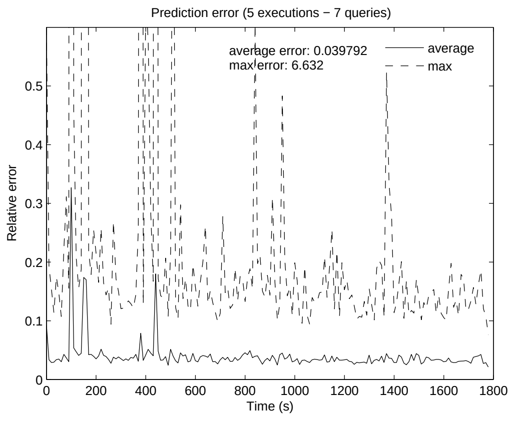

Figure 16: SLR prediction error over time (trace with payloads)

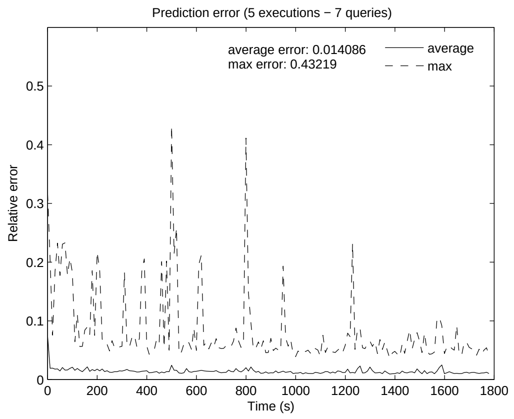

Figure 17: MLR+FCBF prediction error over time (trace with payloads)

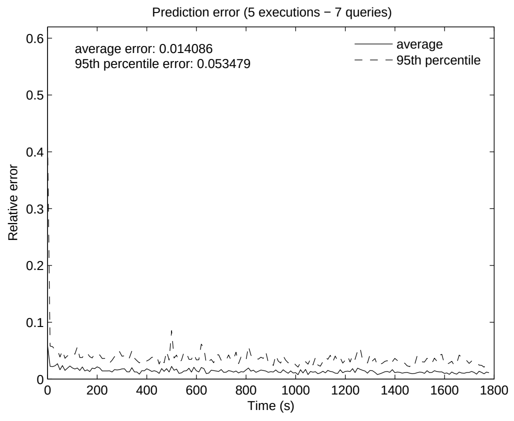

Figure 18: MLR+FCBF 95th-percentile of the error over time (trace with payloads)

our conjecture that EWMA cannot easily adapt to changes in the traffic mix. The second observation is that SLR performs relatively well over all queries for which the number of packets provides enough information on the CPU usage. However, again for *trace* and *string search*, where the packet size matters, it incurs in larger and more variable errors.

6.3 Robustness against Traffic Anomalies

An efficient prediction method for load shedding purposes is most needed in presence of unfriendly traffic mixes. The system may observe extreme traffic conditions when it is monitoring an ongoing denial of service attack, worm infection, or even an attack targeting the measurement system itself.

In order to test our prediction system in this type of traffic conditions, we injected synthetic anomalies into our traces. We have generated many different types of attacks to emulate simple such as volume-based denial of service attacks (i.e., an overwhelming number of packets destined to a single target), worm outbreaks (i.e., a large number of packets from many different source and destinations while keeping the destination port number fixed) or attacks against our monitoring system (i.e., attacks that result in a highly variable and unpredictable workloads to the system).

Figures 19, 20 and 21 show the performance of the three methods in the presence of attacks targeting the monitoring system. We injected in the trace a distributed denial of service attack with spoofed source IP addresses and ports. The attack has additional features aimed at generating a highly variable and difficult to predict workload: the attack goes idle every other second but when idle it still sends the additional packets in a single flow to match existing number of packets in the traffic. The figure shows the performance for the *flows* query that is the most affected by this type of attacks.

In Figure 21, we can see that MLR predictions track very closely the actual CPU usage, with errors around the 10% mark. MLR can anticipate the increase in CPU cycles while EWMA (Figure 21) is always a little behind, resulting in large oscillations in the prediction error. In the case of SLR

	EWMA		*SLR*		*MLR+FCBF*	
Query	Mean	Stdev	Mean	Stdev	Mean	Stdev
application	0.0533	0.0504	0.0254	0.0344	0.0161	0.0179
flows	0.0677	0.0722	0.0579	0.0828	0.0337	0.0335
high-watermark	0.0441	0.0415	0.0077	0.0199	0.0074	0.0139
link-count	0.0429	0.0407	0.0050	0.0161	0.0053	0.0119
popular destinations	0.0504	0.0606	0.0191	0.0386	0.0187	0.0285
string search	0.0748	0.3339	0.0805	0.7953	0.0245	0.0592
trace	0.0563	0.1780	0.0397	0.3579	0.0218	0.0585

Table 5: EWMA, SLR and MLR+FCBF error statistics per query (5 executions)

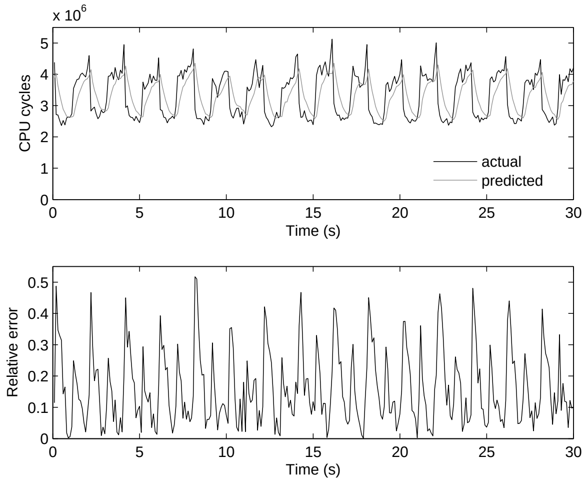

Figure 19: EWMA prediction in the presence of DDoS anomalies (*flows* query)

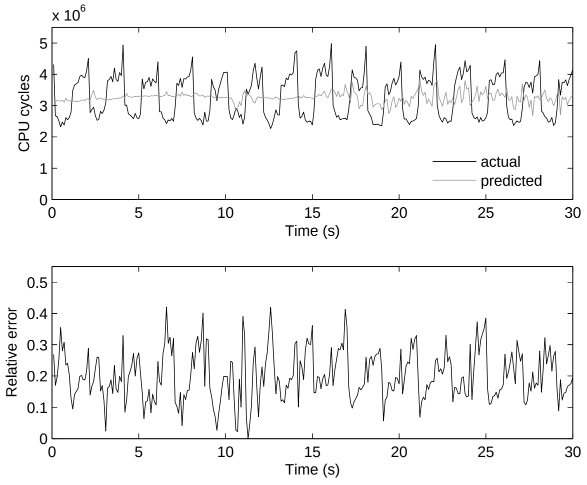

Figure 20: SLR prediction in the presence of DDoS anomalies (*flows* query)

(Figure 20), since the number of packets does not vary as much as the number of 5-tuple flows, the errors are more stable but persistently around 30% (it converges to the average cost per packet between the anomalous and normal traffic).

We also generated other types of attacks that targeted other queries with similar results. For example, we generated an attack consisting of sending burst of 1500 byte long packets for those queries that depend on the number of bytes (e.g. *trace* and *string search*).

In the following experiment, we evaluate instead the impact of this kind of attacks on an actual load shedding implementation based on our prediction method. As an example, we consider a query that tracks the number of active (i.e., for which at least one packet was observed) 5-tuple flows in the packet streams and reports the count every measurement interval. We have implemented that query with a few simple modifications to the *flows* query.

In order to test the effectiveness of our approach, we implement a simple load shedding scheme that makes use of packet and flow sampling. In particular, we manually set a limit $maximum\_cycles$ on the amount of CPU cycles available to the query for processing a batch (set to 4M and 6M cycles per batch in the trace with and without payloads, respectively). The sampling rate to be applied, when the CPU usage is above this threshold, is set to the ratio $maximum\_cycles/predicted\_query\_cycles$.

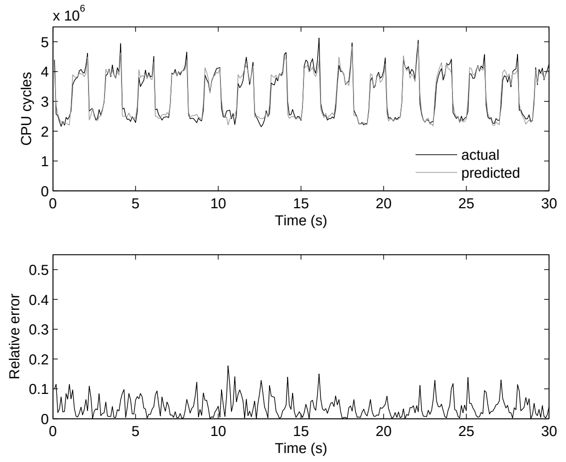

Figure 21: MLR+FCBF prediction in the presence of DDoS anomalies (*flows* query)

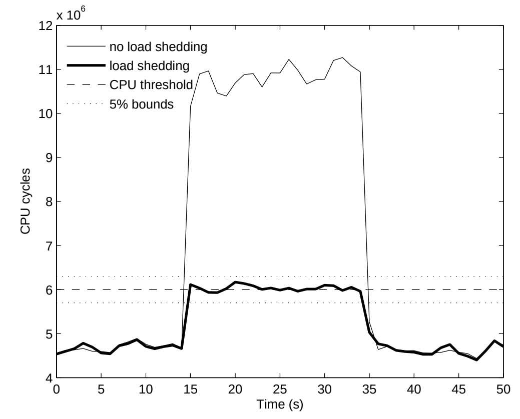

Figure 22: CPU usage with and without load shedding (trace without payloads)

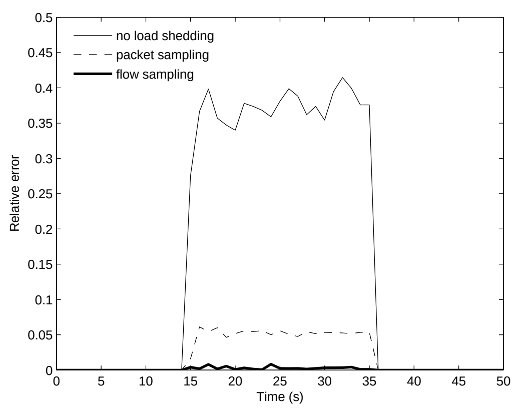

Figure 23: Error in the query results with and without load shedding (trace without payloads)

During 20 seconds (200 batches), we inject a burst of traffic corresponding to a SYN-flood attack with spoofed IP source addresses to force higher CPU usage.

The top plots in Figure 22 shows the evolution of the CPU usage during the anomaly with and without load shedding (with flow sampling) for the trace without payloads. Without load shedding, the CPU cycles increase from 4.5M to 11M cycles during the anomaly (assuming an infinite buffer that causes no packet drops). Instead, when load shedding is enabled, the CPU usage is well under control within a 5% margin of the set target usage.

The bottom plot in Figure 23 shows the query accuracy during the anomaly. To estimate the error in the absence of load shedding, we emulate a system with a buffer of $200ms$ of traffic and 6M cycles available to process incoming traffic. If the CPU usage exceeds 6M, we assume that a queue of packets starts building up until the buffer is full and incoming packets are dropped without control. When load shedding is enabled, the error in the estimation of the number of flows when using flow sampling is less than 1%, while when using packet sampling it is slightly larger than 5%. Without load shedding, the measurement error is in the $35 - 40\%$ range.

As we can observe in Figure 24 and 25, we obtain similar results for the trace with payloads. The slight differences in both figures are due to the way we generated the attack. In particular, we inserted a SYN-flood attack out of every 5 packets. Given that the trace with payloads has less packets than the trace with only packet headers (see Table 2), it results in a less aggressive attack.

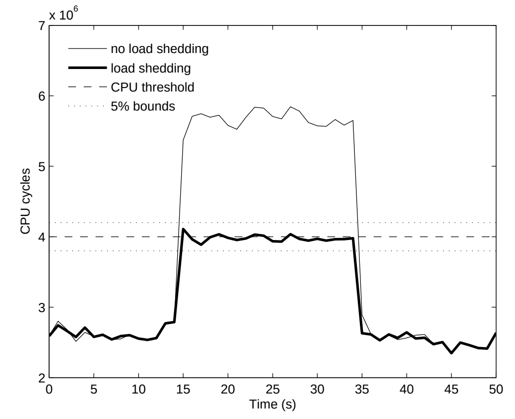

Figure 24: CPU usage with and without load shedding (trace with payloads)

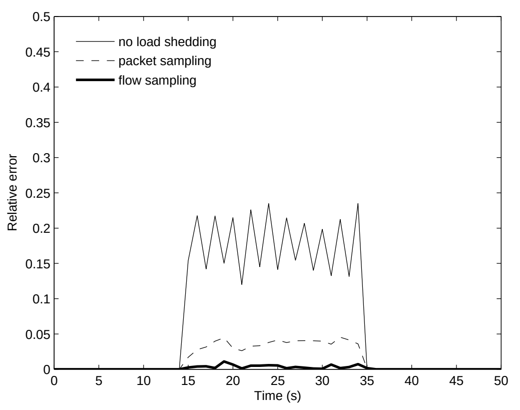

Figure 25: Error in the query results with and without load shedding (trace with payloads)

6.4 Prediction Cost

To understand the cost of running the prediction we compare the CPU cycles of the prediction subsystem to those spent by the entire CoMo process. Table 6 summarizes the results showing the breakdown of the overhead by component.

The feature extraction phase constitutes the bulk of the processing cost. This is not surprising, since several features have to be extracted from every batch (i.e., every $100ms$).

Prediction phase	Overhead
Feature extraction	9.070%
FCBF	1.702%
MLR	0.201%
TOTAL	10.973%

Table 6: Prediction overhead (5 executions)

Furthermore, our current implementation does not interact with the rest of the CoMo system and incurs in additional overhead. An alternative would be to merge the filtering process with the prediction in order to avoid having to scan each packet twice (first to apply the filter and then to extract the features) and to share computations between queries that share the same filter rule. Integrating better the prediction subsystem with the rest of the CoMo platform is part of our on-going work.

The overhead introduced by the FCBF feature selection algorithm is only around 1.7% and the MLR imposes an even lower overhead (0.2%), mainly due to the fact that, when using the FCBF, the number of predictors is significantly reduced and thus there is a smaller number of variables to estimate. The use of FCBF allows to increase the number of features without affecting the cost of MLR.

7 Conclusions and Future work

In this paper, we presented a system that is able to predict the resource requirements of arbitrary and continuous traffic queries without having any explicit knowledge of the computations they perform. It extracts a set of features from the traffic streams to build a prediction model with deterministic (and small) worst case computational cost. We implemented our prediction model in a real system and tested it experimentally with real packet traces.

Our results show that the system is able to predict the resources required to run a representative set of queries with small errors even in the presence of traffic anomalies and attacks targeted against the measurement system itself.

In the paper, we have already identified several areas of future work. In particular, we are currently working on designing and implementing a load shedding scheme that uses the output of our prediction method to guide the system on deciding when, where and how much load to shed. We are also working on implementing the prediction system in a way that is more integrated with the rest of the CoMo platform. We hope that this way we will be able to reduce the overhead of the prediction as well as punctual errors due to the disk accesses. We also intend to extend the set of features we extract to payload-related ones as well. Finally, we are interested in applying similar techniques to other system resources such as memory or disk bandwidth.

Availability

The source code of the prediction and load shedding system described in this paper is available at `http://loadshedding.ccaba.upc.edu`.

Acknowledgments

This work has been founded by a university research grant awarded by the Intel Research Council. Authors also would like to thank the Supercomputing Center of Catalonia (CESCA) for allowing them to collect the packet traces used in this work.

References

[1] Z. Bar-Yossef, T. S. Jayram, R. Kumar, D. Sivakumar, and L. Trevisan. Counting distinct elements in a data stream. In *Proceedings of RANDOM*, 2002.

[2] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7), 1970.

[3] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, 1977.

[4] D. Carney et al. Monitoring streams - a new class of data management applications. In *Proceedings of VLDB*, 2002.

[5] S. Chandrasekaran et al. TelegraphCQ: Continuous dataflow processing of an uncertain world. In *Proceedings of CIDR*, 2003.

[6] Y. Chi, P. S. Yu, H. Wang, and R. R. Muntz. Loadstar: A load shedding scheme for classifying data streams. In *Proceedings of SDM*, 2005.

[7] Cisco Systems. NetFlow services and applications. White Paper, 2000.

[8] C. D. Cranor, Y. Gao, T. Johnson, V. Shkapenyuk, and O. Spatscheck. Gigascope: high performance network monitoring with an sql interface. In M. J. Franklin, B. Moon, and A. Ailamaki, editors, *SIGMOD Conference*, page 623. ACM, 2002.

[9] A. Das, J. Gehrke, and M. Riedewald. Approximate join processing over data streams. In A. Y. Halevy, Z. G. Ives, and A. Doan, editors, *SIGMOD Conference*, pages 40–51. ACM, 2003.

[10] W. R. Dillon and M. Goldstein. *Multivariate Analysis: Methods and Applications*. John Wiley and Sons, 1984.

[11] Endace. http://www.endace.com.

[12] C. Estan, K. Keys, D. Moore, and G. Varghese. Building a better NetFlow. In *Proceedings of ACM Sigcomm*, Aug. 2004.

[13] C. Estan, G. Varghese, and M. Fisk. Bitmap algorithms for counting active flows on high speed links. In *Proceedings of IMC*, 2003.

[14] G. Iannaccone. Fast prototyping of network data mining applications. In *Proceedings of PAM*, Mar. 2006.

[15] Intel. *The IA-32 Intel Architecture Software Developer's Manual, Volume 3B: System Programming Guide, Part 2*. Intel Corporation, 2006.

[16] J. Kang, J. F. Naughton, and S. Viglas. Evaluating window joins over unbounded streams. In U. Dayal, K. Ramamritham, and T. M. Vijayaraman, editors, *ICDE*, pages 341–352. IEEE Computer Society, 2003.

[17] K. Keys, D. Moore, and C. Estan. A robust system for accurate real-time summaries of internet traffic. In *Proceedings of ACM Sigmetrics*, 2005.

[18] NLANR: National Laboratory for Applied Network Research. http://www.nlanr.net.

[19] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, New York, NY, USA, 2nd edition, 1992.

[20] F. Reiss and J. M. Hellerstein. Declarative network monitoring with an underprovisioned query processor. In *Proceedings of ICDE*, 2006.

[21] N. Tatbul, U. Çetintemel, S. B. Zdonik, M. Cherniack, and M. Stonebraker. Load shedding in a data stream manager. In *Proceedings of VLDB*, 2003.

[22] K.-Y. Whang, B. T. Vander-Zanden, and H. M. Taylor. A linear-time probabilistic counting algorithm for database applications. *ACM Trans. Database Syst.*, 15(2):208–229, 1990.

[23] L. Yu and H. Liu. Feature selection for high-dimensional data: A fast correlation-based filter solution. In *Proceedings of ICML*, pages 856–863, 2003.