**RC25353 (CHI1302-002) February 19, 2013**
**Computer Science**

# IBM Research Report

## A Self-Optimizing Workload Management Solution for Cloud Applications

**Haishan Wu[1], Asser N. Tantawi[2], Tao Yu[1]**

[1]IBM Research Division
China Research Laboratory
Building 19, Zhouguancun Software Park
8 Dongbeiwang West Road, Haidian District
Beijing, 100193
P.R.China

[2]IBM Research Division
Thomas J. Watson Research Center
P.O. Box 208
Yorktown Heights, NY 10598
USA

**Research Division**
**Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich**

# A Self-Optimizing Workload Management Solution for Cloud Applications

Haishan Wu
*IBM China Research Laboratory*
*Beijing, China*
*wuhais@cn.ibm.com*

Asser N. Tantawi
*IBM T.J. Watson Research Center*
*Yorktown Heights, NY, USA*
*tantawi@us.ibm.com*

Tao Yu
*IBM China Research Laboratory*
*Beijing, China*
*yutaoyt@cn.ibm.com*

***Abstract*—Given the dynamic nature of the Cloud, resulting from mapping virtual to physical resources, changes in the usage pattern of resources, and migration of virtual resources, in addition to the dynamic nature of the applications themselves, the bottleneck resource in a given application changes over time. Whether the bottleneck resource is a hardware or software resource, physical or virtual, the performance of the application deteriorates accordingly, leading to longer response times and saturated throughput. Today, a typical design of a workload manager for an application manages to a target performance measure, e.g. response time, and/or a system related performance measure, e.g. CPU utilization. We claim that, due to the dynamics of the Cloud and the unavailability of credible resource performance measures, an application workload manager needs to perform its objective in the absence of such target values. Moreover, the application workload manager should be able to function with a model-free controller, thus avoiding the complexity of dynamically changing its model as the cloud environment, especially the bottleneck resource, where the application is hosted and managed changes. In this paper we focus on an application workload manager which uses admission control as a means for load management. We present a design for such a self-optimizing workload manager that is both target-less and model-free. Our approach is to devise black-box bottleneck analytic techniques, combined with a simple binary controller. We demonstrate the validity of our workload manager in an experimental setup, using the RUBiS web application benchmark.**

***Keywords*-cloud computing; workload management; black-box modeling; bottleneck analysis;**

## I. Introduction

Cloud computing introduced PaaS (Platform as a Service) as a new enhanced model for delivering computing and storage as services to end-recipients. In the PaaS model, cloud providers offer a computing platform which typically includes operating systems, programming language execution environments, database and web servers [1]. A multitude of complex applications are hosted by PaaS and, consequently, a subset of resources may get saturated as the load on the system increases, thus causing performance deterioration to applications due to congestion and hence limiting their throughput. The particular resources which get saturated depend on the nature of the applications and their workload. Such resources may be hardware components, such as computing and communication units, or software components, such as server thread pool size, database connections and locks. Further, the saturated resources may change over time depending on the dynamics of the applications, workload types, and the platform configuration. The bottleneck dynamics and migration pattern among the database, application, and clustering middleware tiers are well studied [2], [3], [4].

There are numerous efforts that have addressed the challenges of managing the (cloud) application performance. Most of them focus on managing the bottleneck resource whose type is known or static. Mostly, such work is built on an extremely detailed understanding of the system. Namely, each machine in the system is instrumented with monitors that collect the known type of bottleneck metrics. For example, [5] introduces a way to provision resources for N-tier web applications in the Cloud with the assumption that non-CPU resources are adequately provisioned. Work in [6], [7], [8], [9] assume that either the CPU or the memory is the bottleneck. However, the challenge of bottleneck dynamics, especially in the Cloud, breaks this basic assumption.

Other performance management solutions, such as AzureWatch [10], are built up base on dynamically determine the saturated resources. However, the problem of dynamically determining the saturated resources, and consequently taking corrective actions, such as system reconfiguration and/or admission control, is a complex one. There are several known solutions to such a problem. One approach is to instrument the system with monitors which keep track of the utilization of the various resources and/or other performance metrics (e.g., end to end response time). If the utilization of a given resource approaches a predefined saturation threshold, the associated resource monitor sends a notification through a monitoring infrastructure, identifying the saturated resource and its level of saturation. Accordingly, the management system takes corrective actions based on logic derived from predetermined rules or analytical models [11], [10]. The drawback of this technique is that monitoring agents have to be deployed in the computing system, along with a need for a scalable monitoring infrastructure. This amounts to development cost, run-time performance degradation, and support of various run-time environments. Moreover, it is skill intensive to define the critical threshold for various kinds of resources. In a Cloud environment, this problem becomes even more challenging since the management com-

ponent of a deployed application may only have access to utilization of virtual resources, without knowledge of actual congestion on physical resources. Further, specifying targets on user perceived performance measures, such as response time, is challenging since, on one hand, it is quite a task for an administrator to come up with absolute target values, especially if there is a multitude of different types of requests, and, on the other hand, the physical environment of the application may keep changing by the cloud provider management system, making it difficult to specify reasonable values for such measures.

Another approach is the development of a model of the computing system, which may be analytical, operational, or simulation, in order to identify potential saturated resources by monitoring the various resources in the model. Such a model would have to be continually calibrated given system measurements, such as throughput and response times. Model calibration involves the estimation of parameters in the model through inference/filtering of measured quantities. In [12], a black-box bottleneck analysis mechanism is demonstrated. However, a baseline test is required, which may not be practical. Further, it is not applicable for applications with large variation in service time. Moreover, approaches of this kind suffer from difficulty in dynamically calibrating the model so as to mimic the actual computing system [13].

Meanwhile, workload management has been successfully dealt with in communication system using the TCP protocol in limiting congestion due to bandwidth sharing. Given the amount of available bandwidth and the set of competing users, even thought none of them is known, TCP rapidly, and dynamically, stabilizes its rate into a steady, efficient and fair operating point [14], [15], [16], [17]. Among the available TCP bottleneck detection approaches, an end-to-end, delay-based bottleneck detection, which leverages queueing delay as the congestion measure, has been proposed [18], [19]. This is a preventive approach since delay corresponds to partially filled buffers [20]. When applying such techniques to PaaS systems, one may use measures such as end-to-end average response time, which is the time between sending an http request and receiving its corresponding response.

Inspired by the flow management conducted by TCP, we proposed a workload management solution for cloud applications. We depart from having to (1) specify target, or threshold, values for performance metrics, and (2) build a model of the system which ought to dynamically change as the application environment changes in the Cloud. In particular, we focus on a workload management solution which uses admission control as a means for load management, and design a self-optimizing workload manager that is both target-less and model-free. Our solution consists of two components: (1) a bottleneck analyzer which captures resource saturation at runtime, and (2) a binary controller which regulates the load admission rate to the application. The paper makes two major contributions. First, we present evidences suggesting that the $knee$ of the response time curve is the best operating point for platforms that host web applications. Second, in order to evaluate the performance of the proposed workload manager, we conduct a set of rigorous experiments in the presence of heterogeneous and dynamic environments where flows join and depart asynchronously. The experiments are conducted in a commercially available PaaS, using the RUBiS (Rice University Bidding System) [21] e-commerce web application benchmark. The preliminary experimental results illustrate throughput, fairness, stability, and responsiveness of our solution.

The remainder of the paper is organized as follows. In Section II, we provide background of the platform infrastructure, as well as the admission control which we apply during runtime. The relationship between throughput and response time is explored in Section III and our solution in analyzing resource saturation is proposed. Section IV presents the binary controller design and is evaluated in Section V. Section VI concludes the paper and discusses future work.

## II. Background

### A. System Overview

A typical offering in enterprise-level cloud environments is virtual application [22], which is a collection of application components, behavioral policies, and their links. The virtual application being offered could be of various kinds. As shown in Figure 1, virtual application 1 is a multi-tier enterprise application (e.g., a typical RUBiS benchmark application, or a business intelligence application cluster which consumes data stored in a Relational Database Management System (RDMS)). It is composed of application server, database server and the database JDBC connectivity. Virtual application *n* is a single-tier application which is composed of the application server tier. There is a elastic load balancing service (ELB) [23], which is in charge of dispatching client requests to each of the virtual applications, placed at the entry point of the cloud platform.

Admission Control (AC) is one of the potential actions of the runtime management system in attempting to maintain a good performance for virtual applications. It is accomplished by throttling the workload at the entry of the cloud platform. The target is to make sure that the admitted requests will not saturate any potential bottleneck resource. Admission control may be applied in cases where the saturated resource is unknown due to bottleneck dynamics, or the saturated resource has reached its provisioned threshold limit, or the saturated resource is not scalable, for example, a low configuration of thread pool size.

The AC, which is consumable through service management API, is provided by ELB. Control is managed at the granularity of a virtual application. As depicted in Figure 1, after reception by the proxy, requests are classified into one

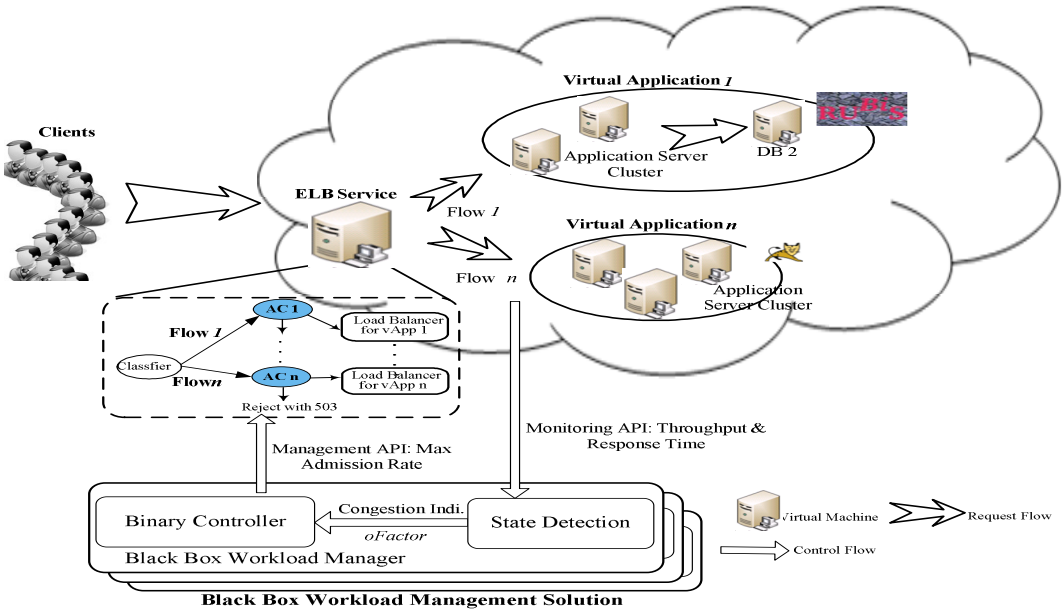



Figure 1. Platform architecture.

of many different flows. We define a flow as the set of requests which belong to the same virtual application and the same service class. The AC is implemented as a rate limiter. It notes the flow of both requests and responses, and allows at most $R$ admitted requests per second. This is done at the session, rather than at the request level. In other words, new sessions may be rejected until the specified $R$ req/sec is achieved. If a session is rejected, its session-initiating request is rejected with a 503 response code. There exists one load balancer per each virtual application. Admitted requests are balanced using a weighted round-robin scheme over the back-end servers. The focus of our work is on the dynamic setting of $R$, in order not to overload or underutilize the available resources for the virtual application.

Each AC possesses its own individual black-box workload manager, which is in charge of setting $R$ through the service management API. Figure 1 illustrates the system architecture for the black-box workload manager. Such an architecture is easy to scale since the controllers work independently of each other. In contrast to some feedback controller designs, this controller operates without any predefined target. In practice, such a target would be hard to set due to bottleneck dynamics. The fed-back measures from the system under control are: "Throughput", which is the rate of successful requests, and "Response Time", which is the time between sending a request and receiving a response. These two metrics are typically obtained through the Monitoring APIs, which are provided either by ELB or the monitoring service. Based on the response time and the throughput variations, the state detection module makes a binary congestion decision. According to the congestion decision, the binary controller adjusts the admission rate by setting $R$. We provide the details of the state detection module and the binary controller in Section III and Section IV, respectively. We use a fixed control cycle of 120 seconds. It is worthwhile to do further investigation on how the state detection together with the controller behavior are affected by the length of the control cycle. This is a topic for future research.

### B. Experimental Setup

A multi-tier virtual application, that is composed of enterprise application and database, is considered. Our testbed utilizes an IBM commercially available PaaS, which provides two Virtual Machines (VM) for ELB, one VM for the application server node, with one application server instance being deployed. We use MySQL as the database server, which is deployed in the 3rd VM. Note that the number of the middleware instances and VMs won't impact the design of the flow controller. The solution being investigated is applicable for virtual application that is hosted by server cluster.

RUBiS [21], which is an auction site prototype modeled after eBay.com, is employed as the application benchmark. The core functionality of an auction site such as selling, browsing, and bidding are implemented. The types of the next request generated by the virtual clients are defined by a state transition matrix that specifies the probability to go from one interaction to another. In our experiment, the default transition table which includes the workloads with read/write interaction mixes is used. Each session is closed loop. That is, a request is generated after receiving the response from the previous request. But the session creation is open loop. The new sessions are created independently of

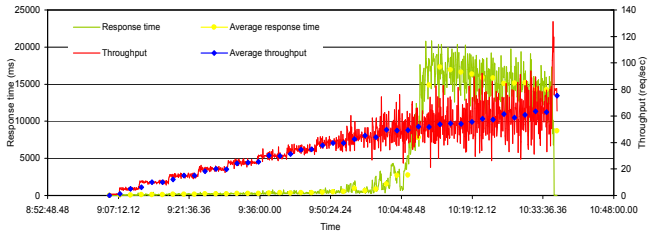



Figure 2. Throughput and response time: reach the bottleneck by slowly ramping up the offered load.

the system's ability to handle them. Load that is offered to the system is adjusted through adjusting the creation pattern for the new session.

## III. Black Box Bottleneck Analysis

In this section, we present the design of the state detection module. Two black box techniques are considered. Both of them can analyze bottlenecks without having monitors deployed in the system. The latter is the one adopted by the workload manager. From the external measurements provided through the monitoring APIs, the first approach attempts to build a pre-defined model of the system, typically a queueing model, by dynamically estimating the parameters of the model. The second approach, which is model-free, works directly with the measurements and derives operational quantities relating the various measures.

### *A. Bottleneck identification of the experimental system by WAIT*

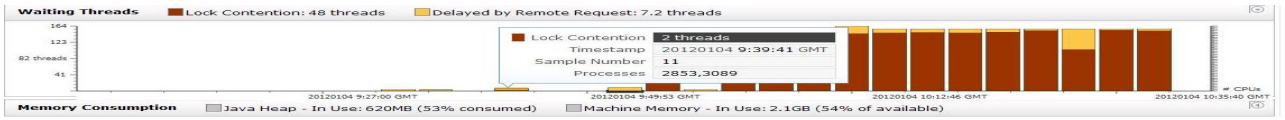
In order to assess the goodness of the techniques to be presented, we first offline identify the bottleneck in the application using a white-box based root cause analysis tool named WAIT [25]. We slowly ramp up the load until a resource bottleneck is reached. WAIT is enabled on the WAS node, with a three-minute sampling interval. Enterprise-class multi-tier applications often suffer from performance problems that manifest as *idle time*, which is indicated by a lack of forward motion [25]. WAIT is good at revealing the root cause of performance issue by detecting the root cause of idle time in a Java tier.

In Figure 2 we present *Throughput* and *Response time* measurements of workload type for which the offered load is ramped up in a step function fashion. They are measured with a 5-second sampling interval. When the load is light, there is no congestion and hence the response time is just the service time needed to process the request. As the load increases, congestion starts to occur and the response time begins to increase. During this phase the $knee$ of this relationship occurs. Finally, in an overload situation, the bottleneck resource limits the throughput of the system and the response time keeps increasing due to a queue buildup for the bottleneck resource.

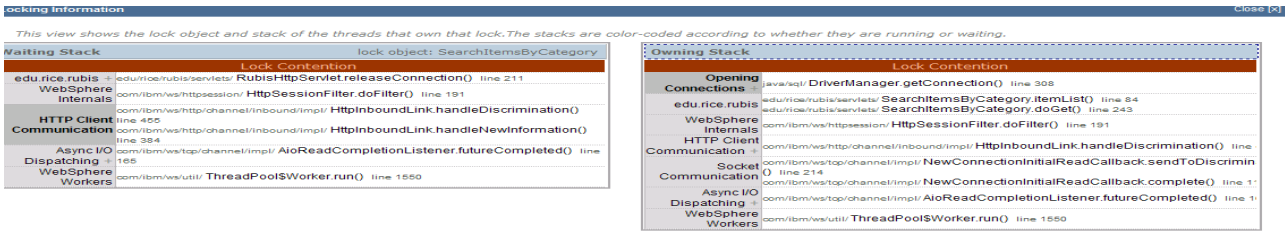
The WAIT report for WAS is depicted in Figure 3(a). The Waiting Threads timeline shows a sudden and sustained surge of *Lock Contention* started at 9:39:41, which is the time when the system started to saturate and the bottleneck queue started to build up. The throughput is around 30 req/sec at this time. The lock contention becomes more and more significant after 9:52:23. While at this point, the throughput reached about 40 req/sec. The *Stack Viewer* in Figure 3(b) suggests that lock contention comes from lacking of WAS to database connections, which may be caused by a misconfigration of the connection pool or the bottleneck issue coming from the database tier. The measurement from WAIT indicates that the best operating point for the experimental systems is between 30 req/sec and 40 req/sec.

### *B. Model-based technique: Black-box modeling*

Building a queueing model of the system is a quite natural approach to addressing resource bottleneck. After all, modeling resource congestion through a queue and server combination is rather straightforward. The server with the highest utilization corresponds to the most likely bottleneck resource. However, this approaches requires knowledge of the resources in the system, their capacities and usage, their allocation and scheduling, as well as the flow of work within the system. A queuing model reflecting the dynamics of the system may consist of a large set of interconnected severs and queues, forming a queueing network. Depending on the complexity of the model, it is solved analytically, numerically, or by simulation. The challenge, after developing such a queueing network model, is to determine the parameters of the model, namely the service times of the servers. A large number of research work dealt exactly with the issue of model parameter estimation. In this paper we focused on a simple queueing model which consists of a single server, representing the bottleneck resource, and a delay, representing the collection of all other resources in the system [13]. The model has been used in a control loop to manage enterprise systems [26]. As such it is an open system, and it has only two parameters to estimate. However, many computing systems exhibit a closed system behavior, where a finite number of users alternate between submitting work, through requests, and waiting for results, through responses. This closed model is also known as a central server model or machine repairmen model. For such closed models, two more parameters are needed: the number of users, also known as population size, and the time to generate a request, known as think time. Hence, a closed model has four parameters to estimate. Given external measurements of throughput and response time, the parameter estimation process typically involves a tracking filter which minimizes the discrepancy between model-based measures and actual measurements, and appropriately adjusting the values of the parameters [13].




(a) Waiting threads.




(b) Stack viewer for lock contention.

Figure 3. Root cause analysis: WAIT report for WAS.

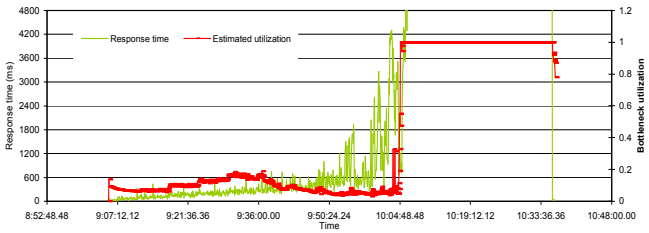



Figure 4. Model based: Estimate the bottleneck utilization.

We used the measurements of the above experiment to dynamically build the simple queueing model described above and estimate its parameters. The model is then used to obtain measures such as server utilization. The higher the server utilization the more likely a bottleneck condition occurred. The *Throughput* and *Response time* in Figure 2 are chosen as the input for the model, in order to well train the model with data that contain the most information. The results of the model building and estimation is illustrated in Figure 4. A slowly increase of the bottleneck resource utilization is expected due to the slowly ramp up of the offerload. However, the estimated utilization of the server, which corresponds to the bottleneck resource, initially increases to about 20% then decreases to less than 10%, but then eventually, and suddenly, reaches 100%. This indicates a failure in the parameter estimation process. The reason for this misbehavior is that the scales of the service time of the bottleneck resource (server) and all other resources (delay) are order of magnitudes apart. Further, there is a degree of freedom between the choice of values for the population size and think time as noted in [13]. Therefore, we concluded that this approach is not robust enough, as employed, to tackle generally unknown and unpredictable system enviromemts.

### C. Model-free technique: Black-box operational

In lieu of building a model for the system under consideration, this approach builds on simple operational relationships among the external measurements of the black-box system. Further discussion on model-based and model-free approaches may be found in [20]. We filter the measurements in Figure 2 by averaging the samples with a sliding window of size 24 and lag 24. That is an interval of 120 seconds, which is just the control cycle, being introduced to each filtering output. The results are named *Average throughput* and *Average response time* respectively. Dividing the first by the second leads to a well-known measure called $Power$ [27]. (In general, one of the two measures may be raised to a power.) Maximizing the value of $Power$, as an objective, leads to an operating point where throughput is high while the response time is small. For the experiment described above we plot the value of $Power$ in Figure 5. Above it and along the same timeline we show the number of waiting threads. By comparing the progress of the two measures, one notes that $Power$ reaches its limit at the point that roughly corresponds to the knee of the response time - throughput curve, then gradually decreases as the number of waiting threads increases. At that point the bottleneck situation intensifies since the thread queue starts to build up. This suggests that identifying the best operating point of the system can be done through detecting the point where $Power$ is maximum.

When the offered load keeps increasing, only if a decrease in $Power$ is detected, can we claim that the system starts to saturate. In order to detect a decrease in $Power$, we define the ratio of $Power$ as

$$ratio_k = \frac{power_k}{maxPower_k}, \quad (1)$$

where $power_k$ is the $Power$ during cycle $k$ and $maxPower_k$ is the maximum of $Power$ up to and including cycle $k$. Once $ratio_k$ becomes lower than a predefined $RatioThreshold$, which is 1 theoretically, we consider the system to be not well utilized, either because of under utilization or over utilization. Another metric named $bestRespTime_k$, defined as the average response time

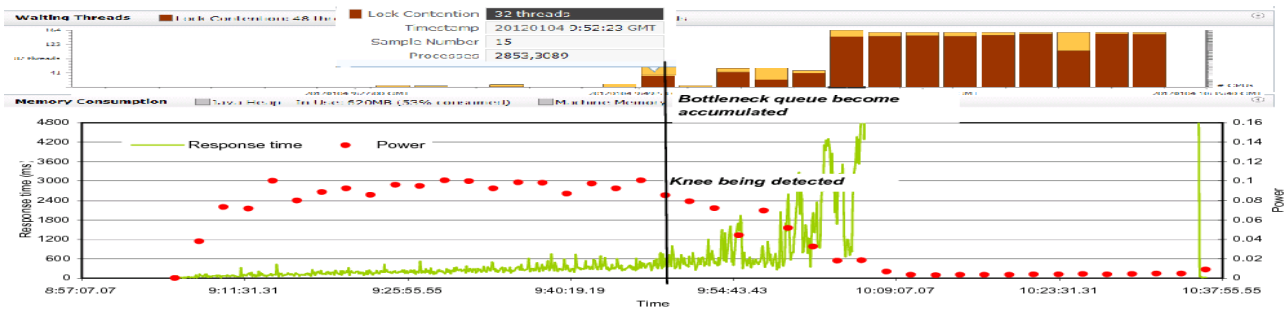



Figure 5. The knee of response time curve matches the system bottleneck.

**Algorithm 1 Congestion detection per each control cycle**

**Inputs**
$averageRespTime$:Average response time between the sampling interval,
$averageThroughput$:Average throughput between the sampling interval.
**Algorithm**
1 : $power = averageRespTime/averageThroughput$
2 : **if** $power > maxPower$ **then**
3 : $maxPower = power$
4 : $bestRespTime = averageRespTime$
5 : **end if**
6 : $ratio = power/maxPower$
7 : $oFactor = 0$
8 : **if** $ratio < RatioThreshold$ &
9 : $averageRespTime > bestRespTime$ **then**
10: $oFactor = 1$
11: **end if**

($averageRespTime_k$) when $maxPower_k$ is achieved, is introduced. $bestRespTime_k$ could be considered as the best expected average response time for that specific application. It could also be considered as a criterion to distinguish between underload and overload, given $ratio_k$ is below $RatioThreshold$.

Following the above discussion, we introduce a heuristic algorithm, outlined in Algorithm 1, which determines the state of the system under control, periodically. The result is binary, with either $oFactor_k = 0$ as underload, or $oFactor_k = 1$ indicating that some of the resources are saturated.

A $RatioThreshold$ of 0.9 is chosen in case of false positive detection in order to accommodate for the noise of measurements. We note from Figure 5 that the knee of the response time - throughput curve, should be detected at around 9:48:43 – 9 minutes after the first lock contention which is detected by WAIT and 4 minutes before the lock contention became significant. Thus, the Black-box operational approach seems effective in detecting/predicting the potential resource saturation without detailed knowledge of the resource bottleneck. In the next section we describe our control law which reacts to the congestion feedback. It is an optimal controller, in the sense that it attempts to estimate the value of $Power$ and manages to keep the system at the maximum of $Power$.

## IV. Controller Design

In this section, we present the design of our binary feedback controller for black-box workload management. We choose the additive increase/multiplicative-decrease (AIMD) algorithm [19], [28], which is a feedback control algorithm best known for its use in TCP Congestion Avoidance, as our binary controller due to its simplicity and robustness. This section focuses on applying the binary feedback control methodology in dealing with bottleneck dynamics in middleware. In this regard, we elaborate on our experience in evaluating the controller, which is especially important in managing the workload, both in this and the following sections.

The AIMD algorithm operates according to the following control law

$$r_k = \begin{cases} r_{k-1} + iFactor & \text{if } oFactor == 0, \\ r_{k-1}/dFactor & \text{if } oFactor == 1, \end{cases} \quad (2)$$

where $r_k$ is the current admission rate during cycle $k$ and $r_{k-1}$ is the sending rate for the previous control cycle. AIMD has two phases: the increment phase which increase the admission rate additively through the increase factor $iFactor$ and the decrement phase which decrease the admission rate multiplicatively through the decrease factor $dFactor$.

Usually, *fairness* and *convergence* are two major criteria for measuring a controller's performance. *fairness* is always used to determine whether users or application are receiving a fair share of system resources. In this paper, we assume that there is no service differentiation among flows that share the same platform. So *fairness*, in our case, means that flows that share the same resource bottleneck should converge to the same admission rate. *convergence* is a combination of *responsiveness*, which is the speed with which the goal state from any starting state is reached, and *smoothness*, which is the magnitude of the oscillations around the optimal state.

Given the binary nature of the feedback, the controller does not generally converge to a single steady state [19]. However, we would expect a controller with good *convergence* should have small *responsiveness* and less oscillation.

The choice of $iFactor$ and $dFactor$, must be set to take the desired control performance into consideration. An experimental research is done to reveal how *fairness* and *convergence* are effected by the choice of $iFactor$ and $dFactor$. We conducted a simple numerical simulation where we simulated three flows that share the same (bandwidth) bottleneck. Each flow has its own individual AIMD controller that reacts to the indication of congestion. The maximum bandwidth is assumed to be 50 req/sec. That is to say, given that the sum of the three flow rates is larger than the maximum bandwidth, each flow will be marked as congested and their AIMD controller will start multiplicative decreasing the request rate. Otherwise, the rate will be increased additively. Based on this simulation experiment, four tests are performed with different configurations for the *increase factor*, *decrease factor*, and the initial admission rate. The configuration for the tests are outlined by Table I. Figure 6 plots the flow rates for all three flows in all tests. Test 1 is done with three flows using the same *increase factor* and *decrease factor*, but with different initial request rates and times when flows start to join the system. According to Figure 6(a), each flow finally converged to around 17 req/sec. Moreover, an equal share of the bandwidth is found since all the flows finally synchronized. Test 1 reveals that the convergence and the fairness of AIMD are oblivious to the initial rate and the starting time. Test 2 describes another three flows that are different from either the *increase factor* or the *decrease factor*. As shown in Figure 6(b), even though convergence is reached for each flow, all the flows ended up with an unfair share of the bandwidth. The comparison between Test 1 and Test 2 reveals that in order to gain a fair share, all the flows need to share the same *increase factor* and *decrease factor*. Test 3 and Test 4 are tests with either a larger *increase factor* or a larger *decrease factor*, when compared to Test 1. Heavier oscillation but faster speed to reaching steady state are observed in Figure 6(c) and Figure 6(d). The comparison between Test 1, Test 3 and Test 4 indicates the trend in *convergence*. Given the simulation results, we decided to choose 2 and 1.1 as the *increase factor* and *decrease factor*, respectively, and move on to the real system verifications.

## V. Results

In this section we present results from three real-world experiments that explore the major objectives of the black-box workload management: capturing the best operation point without the detail knowledge of bottleneck, given the workload dynamics. A sine wave arrival workload type, which is created by varying the number of concurrent users in the load driver, is employed. As illustrated in Figure 7, the

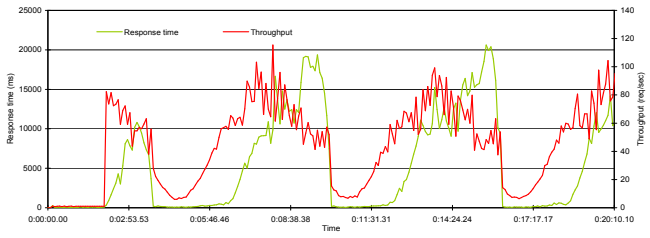



Figure 7. Sine wave arrival: Throughput and response time without control.

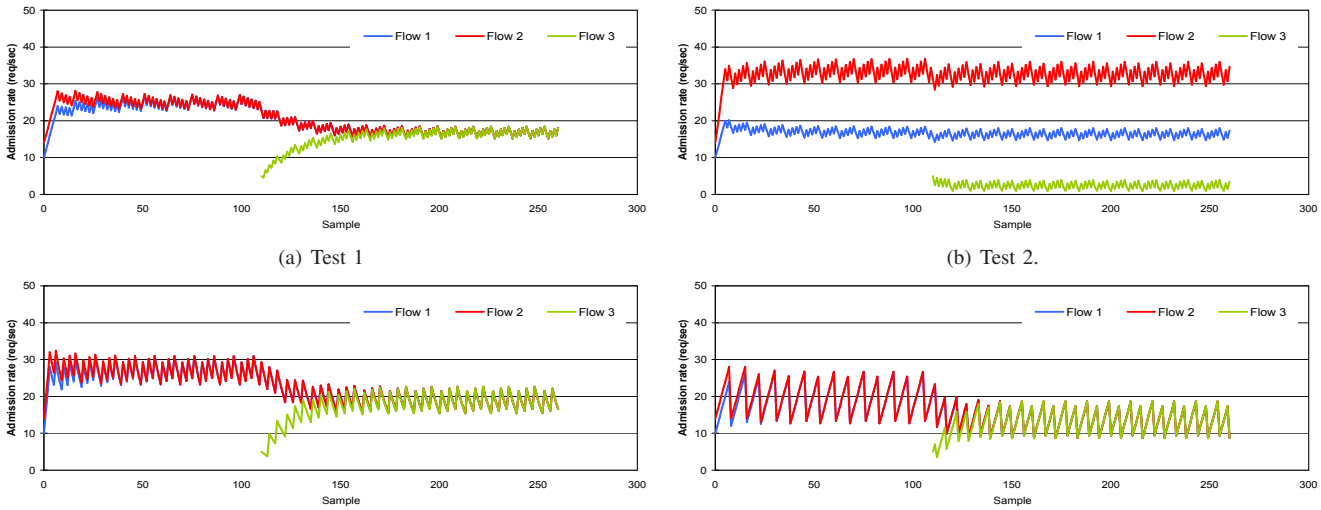
workload keeps varying over time. The response time graph in Figure 7 demonstrates that the system keeps oscillating between underload and overload. Requests that are generated by the same load driver are classified into the same flow. The first scenario is a single flow experiment. The system resources are consumed by one flow without any competition or background consumption by others. The second scenario has two in-phase flows that compete for the same system resources. Each flow is regulated by its individual workload manager without any awareness of the other. In the middle of the experiment, one of the flows leaves the system. This experiment is designed to evaluate the fairness and the adaptability of the controller. The third scenario is quite challenging. Two out-of-phase flows are generated to compete for the same system resources. The robustness of the controller is further evaluated under different workload types. As revealed by the WAIT report in Section III, the best operating point for our experimental system is between 30 req/sec and 40 req/sec. The performance of the controllers are evaluated by whether the best operating point is well tracked with little throughput and response time oscillation.

### A. Scenario 1: Single Flow

In this scenario, the system resources are consumed by a single flow, with one black-box workload manager operating during the whole run. The "Admission rate" in Figure 8 shows the max admission rate regulated by the controller. It only applies to the session-initiating request. Throughput and response time are shown respectively. Even though the controller has little knowledge about the bottleneck, the throughput is able to quickly converge at around 32 req/sec during every ramp up and down cycle, except when the request arrival rate is too low to reach the max admission rate. Very little oscillation exists in the throughput as well as the response time curve, except the time when the workload started to ramp up. Considering the binary nature of the bottleneck indicator (*oFactor*) and the dynamics of both the workload rate and the workload type, the size of the oscillation is considered to be small.

### B. Scenario 2: Two In-Phase Flows

In this scenario, we introduce two in-phase flows that compete for the same system resource. Two RUBiS load drivers are employed to generate the workload type of




(a) Test 1

(b) Test 2.

(c) Test 3.

(d) Test 4.

Figure 6. The impact of the iFactor and the dFactor on controller behavior

Table I
AIMD PARAMETERS.

| **Test** | **Test 1** | | | **Test 2** | | |
|---|---|---|---|---|---|---|
| **Flow** | Flow 1 | Flow 2 | Flow 3 | Flow 1 | Flow 2 | Flow 3 |
| **Increase Factor** | 2 | 2 | 2 | 2 | 4 | 2 |
| **Decrease Factor** | 1.1 | 1.1 | 1.1 | 1.1 | 1.1 | 2 |
| **Init Rate(req/sec)** | 10 | 14 | 5 | 10 | 14 | 5 |
| **Test** | **Test 3** | | | **Test 4** | | |
| **Flow** | Flow 1 | Flow 2 | Flow 3 | Flow 1 | Flow 2 | Flow 3 |
| **Increase Factor** | 6 | 6 | 6 | 2 | 2 | 2 |
| **Decrease Factor** | 1.1 | 1.1 | 1.1 | 2 | 2 | 2 |
| **Init Rate(req/sec)** | 10 | 14 | 5 | 10 | 14 | 5 |

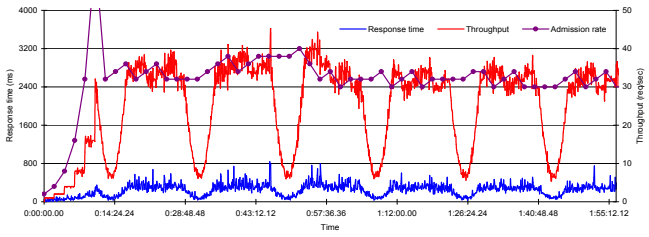



Figure 8. Single flow: Throughput and response time under control.

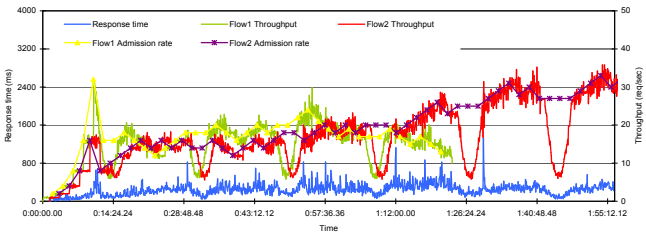
sine wave arrivals. The two sine waves are in-phase, i.e. they synchronously ramp up and down. By comparing with Scenario 1, the speed to overload the system is much faster. Another challenge for Scenario 2 is that each flow is regulated by its individual workload manager, without any awareness of the other. The congestion indication is carried out by the response time, which is shared by the two flows.

As there is no service differentiation between the two flows, it is expected that the system resources be equally divided among them. Jain's fairness index is used to determine whether the two flows are receiving a fair share of system resources. It is given by

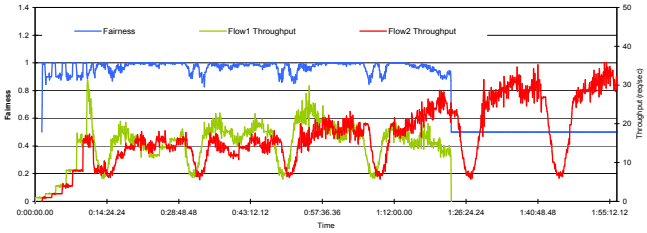
$$f(x_1, x_2, ..., x_n) = \frac{(\sum_{i=1}^{n} x_i)^2}{n \sum_{i=1}^{n} x_i^2} \quad (3)$$

where $n$ is the number of flows and $x_i$ is the throughput for the $i$th flow. The value of $f(x_1, x_2, ..., x_n)$ ranges from $\frac{1}{n}$, which is the worst case, to 1, which is the best case. It is maximized when all flows share the same throughput [27].

The max admission rate regulated by each of the controller and the throughput for each flow are plotted in Figure 9(a), respectively. Moreover, only the response time for flow 2 is shown in Figure 9(a). Flow 1 share the same response time with flow 2, since both flows have similar request types and share the same system resources. Flow 1 and Flow 2 synchronously ramp up and down from the beginning of the experiment. At this time, the sum of the admission rates oscillates between 27 req/sec and 36 req/sec during the steady state for each ramp up and down cycle, which is actually a good convergence given the bottleneck dynamics that are being introduced by the other controller. Eighty minutes after the test started, flow 1 leaves the system. The state detector for flow 2 is able to detect the change of

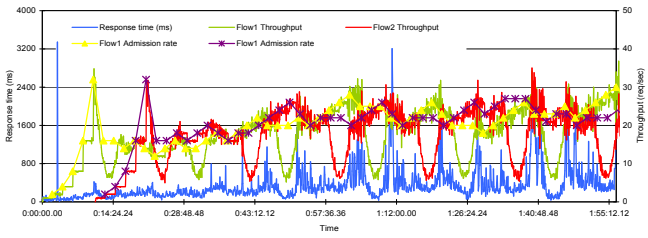



(a) Two flows in phase: Throughput and response time under control.

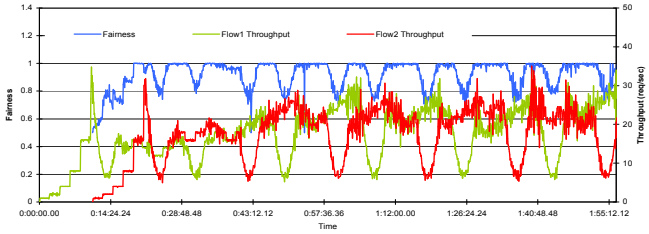



(b) Two flows in phase: Fairness.

Figure 9. Two flow in phase




(a) Two flows opposite phase: Throughput and response time under control.




(b) Two flows opposite phase: Fairness.

Figure 10. Two flow opposite phase

the bottleneck and consequently takes action by ramping up the admission rate to reach another higher operating point. During the last two ramp up and down cycles, flow 2 finally converges to around 32 req/sec with little oscillation.

Fairness is calculated by equation 3 and is plotted in Figure 9(b) together with the throughput for each flow. At the time when both flows coexist, fairness fell below 1 at times. However, that is just the time when the arrival rate started to ramp up. After that, a quick converge to 1 is observed which indicates a good fair share of the bottleneck resources.

### C. Scenario 3: Two Out-of-Phase Flows

In this scenario, two flows opposite in phase, which is the most complex and dynamic workload, are introduced. Similar to Scenario 2, each flow has its individual, autonomous workload manager. The variability of this mixed workload imposes big challenges for detecting the system bottleneck, together with shaping the workload. As shown in Figure 10(a), every ramp up of either flow easily breaks the steady state of the controller. This leads to a transient spike in both throughput and response time. However, both of the controllers are able to regulate the max admission rate at around 20 req/sec, with the sum of the max admission rate at around 40 req/sec. Although the sum of the max admission rate finally converged to the point which is a little bit higher than the best system operating point, it is encouraging to see a good steady state being maintained by each controller. Moreover, the fairness plotted by Figure 10(b) indicates that a fairness of 1 is maintained except at times when one of the flows could not drive enough workload to compete for the resources.

## VI. Conclusions and Future Work

In this paper, we have implemented and validated a self-optimizing, delay-based, black-box workload management solution for cloud applications. Specifically, the workload manager is both target-less and model-free, hence it does not need much configuration nor continual model adjustment to match Cloud dynamics. A black-box bottleneck analysis mechanism that is built on simple operational relationships among the external measurements of the black-box system, is introduced to identify an optimal point where the system should operate. The output of the bottleneck analysis drives the controller behavior in order to react to the highly dynamic workload in a Cloud environment. The black-box workload management solution built up on a commercially available PaaS. It is evaluated under a variety of workloads with time-varying throughput and background bottleneck consumption. Our experimental results show that the black-box workload manager reacts to various workload changes, selecting optimal or close to optimal session-initiating request rate limits. Moreover, a fairness share of the bottleneck between two individual manager is also demonstrated. Such solution is easy to scale since multiple controllers work independently of each other.

We believe that this is the first time that the flow management in TCP is employed in cloud application performance management. Further work is needed to verify the robustness of the controller given the bottleneck shifting among tiers. Further, the error or exception response is another kind of feedback that indicates congestion. Considering the abnormal response is a good enhancement for the black-box bottleneck analysis. Considering the service differentiation, the utility function needs to be involved in designing the increase factor and the decrease factor. Finally, we do

foresee a promising combination between the black-box based bottleneck analysis and the white-box based analysis. The black-box based approach is considered as a health detector which is in charge of selecting the application with potential performance bottleneck. The white-box based root cause analysis tool could be used to find the root case with the least performance overhead. The amount of data to be analyzed is dramatically decreased this way.

## ACKNOWLEDGMENTS

The authors are thankful to Keith Smith for motivating this work and for his comments and suggestions. We are also thankful to Renshi Luo for his help in preparing the background information and the experimental environment.

## REFERENCES

[1] "Microsoft Azure Platform," http://www.microsoft.com/windowsazure/.

[2] C. Amza, E. Cecchet, A. Ch, A. L. Cox, S. Elnikety, R. Gil, J. Marguerite, K. Rajamani, and W. Zwaenepoel, "Bottleneck characterization of dynamic web site benchmarks," Tech. Rep., 2002.

[3] S. Malkowski, M. Hedwig, and C. Pu, "Experimental evaluation of n-tier systems: Observation and analysis of multi-bottlenecks," in *IISWC*, 2009, pp. 118–127.

[4] Q. Wang, S. Malkowski, D. Jayasinghe, P. Xiong, C. Pu, Y. Kanemasa, M. Kawaba, and L. Harada, "The impact of soft resource allocation on n-tier application scalability," in *IPDPS*, 2011, pp. 1034–1045.

[5] P. Xiong, Z. Wang, S. Malkowski, Q. Wang, D. Jayasinghe, and C. Pu, "Economical and robust provisioning of n-tier cloud workloads: A multi-level control approach," in *ICDCS*, 2011, pp. 571–580.

[6] T. F. Abdelzaher and C. Lu, "Modeling and performance control of internet servers," in *Proceedings of the 39th IEEE Conference on Decision and Control, Sydney, Australia*, 2000, pp. 2234–2239.

[7] Y. Diao, N. Gandhi, J. L. Hellerstein, S. Parekh, and D. M. Tilbury, "Using MIMO feedback control to enforce policies for interrelated metrics with application to the Apache web server," in *Network Operation and Management Symposium*, Florence, Italy, April 2002, pp. 219–234.

[8] G. Pacifici, W. Segmuller, M. Spreitzer, and A. Tantawi, "Cpu demand for web serving: Measurement analysis and dynamic estimation," *Performance Evaluation*, vol. 65, no. 6–7, pp. 531–553, June 2008.

[9] Y. Diao, X. Hu, A. N. Tantawi, and H. Wu, "An adaptive feedback controller for sip server memory overload protection," in *ICAC*, 2009, pp. 23–32.

[10] "AzureWatch," http://www.paraleap.com/azurewatch.

[11] R. Scott and D. Skeen, *Application Monitoring with ITCAM for Response Time*, http://www.scribd.com/doc/57820998/Application-Monitoring-With-ITCAM-for-Response-Time.

[12] M. Ben-Yehuda, D. Breitgand, M. Factor, H. Kolodner, V. Kravtsov, and D. Pelleg, "Nap: a building block for remediating performance bottlenecks via black box network analysis," in *ICAC*, 2009, pp. 179–188.

[13] D. Kumar, A. Tantawi, and L. Zhang, "Estimating model parameters of adaptive software systems in real-time," in *Runtime Models for Self-managing Systems and Applications*, ser. Autonomic Systems, D. Ardagna and L. Zhang, Eds. Springer, 2010, pp. 45–71.

[14] V. Jacobson, "Congestion avoidance and control," in *Symposium proceedings on Communications architectures and protocols*, ser. SIGCOMM '88. New York, NY, USA: ACM, 1988, pp. 314–329. [Online]. Available: http://doi.acm.org/10.1145/52324.52356

[15] M. Mathis, J. Mahdavi, S. Floyd, S. Floyd, and A. Romanow, "Tcp selective acknowledgment options," 1996.

[16] V. Jacobson, R. Braden, and D. Borman, "Tcp extensions for high performance," United States, 1992.

[17] D. X. Wei, S. Member, C. Jin, S. H. Low, S. Member, and S. Hegde, "Fast tcp: motivation, architecture, algorithms, performance," in *IEEE Infocom*, 2004.

[18] R. Jain, "A delay-based approach for congestion avoidance in interconnected heterogeneous computer networks," *ACM COMPUTER COMMUNICATION REVIEW*, vol. 19, pp. 56–71, 1989.

[19] D.-M. Chiu and R. Jain, "Analysis of the increase and decrease algorithms for congestion avoidance in computer networks," *Comput. Netw. ISDN Syst.*, vol. 17, no. 1, pp. 1–14, June 1989. [Online]. Available: http://dx.doi.org/10.1016/0169-7552(89)90019-6

[20] J. Martin, A. Nilsson, and I. Rhee, "Delay-based congestion avoidance for tcp," *IEEE/ACM Trans. Netw.*, vol. 11, no. 3, pp. 356–369, June 2003. [Online]. Available: http://dx.doi.org/10.1109/TNET.2003.813038

[21] "RUBiS," http://rubis.ow2.org/.

[22] B. McChesney, J. Bohn, and J. Kochuba, "Manage application services with virtual application patterns," http://www.ibm.com/developerworks/cloud/library/cl-puresystem-vap/index.html?ca=drs-/.

[23] "Elastic Load Balancing," http://aws.amazon.com/elasticloadbalancing/.

[24] IBM, *WebSphere Virtual Enterprise*, http://www.ibm.com/software/webservers/appserv/extend/virtualenterprise/, 2009.

[25] E. A. Altman, M. Arnold, S. Fink, and N. Mitchell, "Performance analysis of idle programs," in *OOPSLA*, 2010, pp. 739–753.

[26] A. Dubey, R. Mehrotra, S. Abdelwahed, and A. Tantawi, "Performance modeling of distributed multi-tier enterprise systems," *SIGMETRICS Perform. Eval. Rev.*, vol. 37, no. 2, pp. 9–11, October 2009.

[27] R. Jain, D.-M. Chiu, and W. Hawe, "A quantitative measure of fairness and discrimination for resource allocation in shared computer systems," *CoRR*, vol. cs.NI/9809099, 1998. [Online]. Available: http://dblp.uni-trier.de/db/journals/corr/corr9809.html#cs-NI-9809099

[28] W. Stevens, "Tcp slow start, congestion avoidance, fast retransmit, and fast recovery algorithms," United States, 1997.