Input Shaping to Achieve Service Level Objectives in Cloud Computing Environments

Submitted in partial fulfillment of the requirements for

the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

Andrew J. Turner

B.S., Computing and Management, Loughborough University
M.S., Computer Science, University of Oxford

Carnegie Mellon University
Pittsburgh, PA

December, 2013

Ph.D. Thesis Committee

Professor Hyong S. Kim, Chair (Carnegie Mellon University)

Professor Raj Rajkumar (Carnegie Mellon University)

Professor James C. Hoe (Carnegie Mellon University)

Dr. Tina Wong (Google)

Keywords: cloud computing, virtualization, resource contention, quality of service, load balancing, resource management, admission control

Acknowledgements

There are more people that I have met at CMU I would like to thank than I could possibly mention. I would first like to thank my advisor Prof. Hyong S. Kim. Without him pushing me I would have never made it this far or completed my PhD. I have learnt as much from him about business and academia over the years as I have about how to be a researcher. I would also like to thank the members of my thesis committee for their input and help; Prof. James C. Hoe, Prof. Raj Rajkumar, and Dr. Tina Wong. I am lucky to have worked with such world-class researchers.

I would like to thank all of my coworkers in the research group for their help over the years. Without their input and feedback my work would have been a much more difficult undertaking. I would like to thank Andrew Fox for also staying up, and giving me someone to talk to during the sleepless days and nights of programming. I would like to thank Dr. Akkarit Sangpetch for fixing so many problems before I even realized that they existed.

I would like to thank all of the friends that I have met over my years in Pittsburgh. The people are the thing I will remember most about my time in Pittsburgh and at CMU. I would like to thank Dr. James Weimer and Dr. Stephen Tully for their valuable insight into local culture.

I would like to thank my parents Ann Turner and Ivan Turner for all of their support and belief in me over the years. I would have not made it this far without their encouragement. I would lastly like to thank my beautiful wife Dr. Beth Foreman. The acknowledgments section isn't large enough for me to describe what a help you have been and the amount of support you have given me.

This thesis was supported in part by Information and Communication Technologies Institute (ICTI), National Science Foundation (NSF) award 0756998, and ARO DAAD19-02-1-0389 and W911NF-09-1-0273.

Abstract

In this thesis we propose a cloud Input Shaper and Dynamic Resource Controller to provide application-level quality of service guarantees in cloud computing environments. The Input Shaper splits the cloud into two areas: one for shaped traffic that achieves quality of service targets, and one for overflow traffic that may not achieve the targets. The Dynamic Resource Controller profiles customers' applications, then calculates and allocates the resources required by the applications to achieve given quality of service targets. The Input Shaper then shapes the rate of incoming requests to ensure that the applications achieve their quality of service targets based on the amount of allocated resources.

To evaluate our system we create a new benchmark application that is suitable for use in cloud computing environments. It is designed to reflect the current design of cloud based applications and can dynamically scale each application tier to handle large and varying workload levels. In addition, the client emulator that drives the benchmark also mimics realistic user behaviors such as browsing from multiple tabs, using JavaScript, and has variable thinking and typing speeds. We show that a cloud management system evaluated using previous benchmarks could violate its estimated quality of service achievement rate by over 20%.

The Input Shaper and Dynamic Resource Controller system consist of an application performance modeler, a resource allocator, decision engine, and an Apache HTTP server module to reshape the rate of incoming web requests. By dynamically allocating resources to applications, we show that their response times can be improved by as much as 30%. Also, the amount of resources required to host applications can be decreased by 20% while achieving quality of service objectives. The Input Shaper can reduce VMs' resource utilization variances by 88%, and reduce the number of servers by 45%.

Contents

List of Figures

List of Tables

1. Introduction

During the past decade cloud computing has become an increasingly popular environment for businesses to host their applications. Recent surveys show that over 50% of businesses currently use at least one cloud computing provider [1]. Cloud computing provides business customers with Infrastructure-as-a-Service. This allows customers to only pay for the computing resources that they consume, rather than paying the cost of running their own datacenter [2]. This is desirable as the average resource utilization of a physical server is only 5%-20% [3]. Cloud computing providers run customers' applications inside of virtual machines (VMs) on shared physical servers known as hosts. Placing multiple applications per host allows the average resource utilization to be higher and reduces the number of hosts required to run all of the customers' applications. This in turn reduces the cost of purchasing and maintaining the hosts, allowing cloud computing providers to offer computing resources at an attractive price.

Figure 1-1 shows a simplistic overview of a cloud computing environment. The cloud itself is a collection of physical hardware – servers, network switches, disk storage, etc. – much the same as a traditional datacenter. Rather than running applications on 'bare metal' physical servers, cloud environments virtualize their available resources to share them amongst their customers. This is achieved using a VM Hypervisor such as VMware ESX [4] or Xen [5]. Customers can request a VM, which the cloud creates and places on a shared host. Customers do not get to choose which hosts their VMs are located on as they do not have direct access to

the underlying hardware. The cloud administrator can choose where customers' VMs are located as they do have access to the underlying hardware and can control the Hypervisor. However, as a cloud computing environment can contain millions of VMs, the placement of VMs is actually performed by an automated management system. The management system chooses the location of each VM based upon certain goals. A typical goal is attempting to maximize resource utilization [6]. Lastly, we have the application users or clients. The users access applications that are placed in the cloud by the cloud's customers. Examples of common cloud based applications are websites or video streaming services.

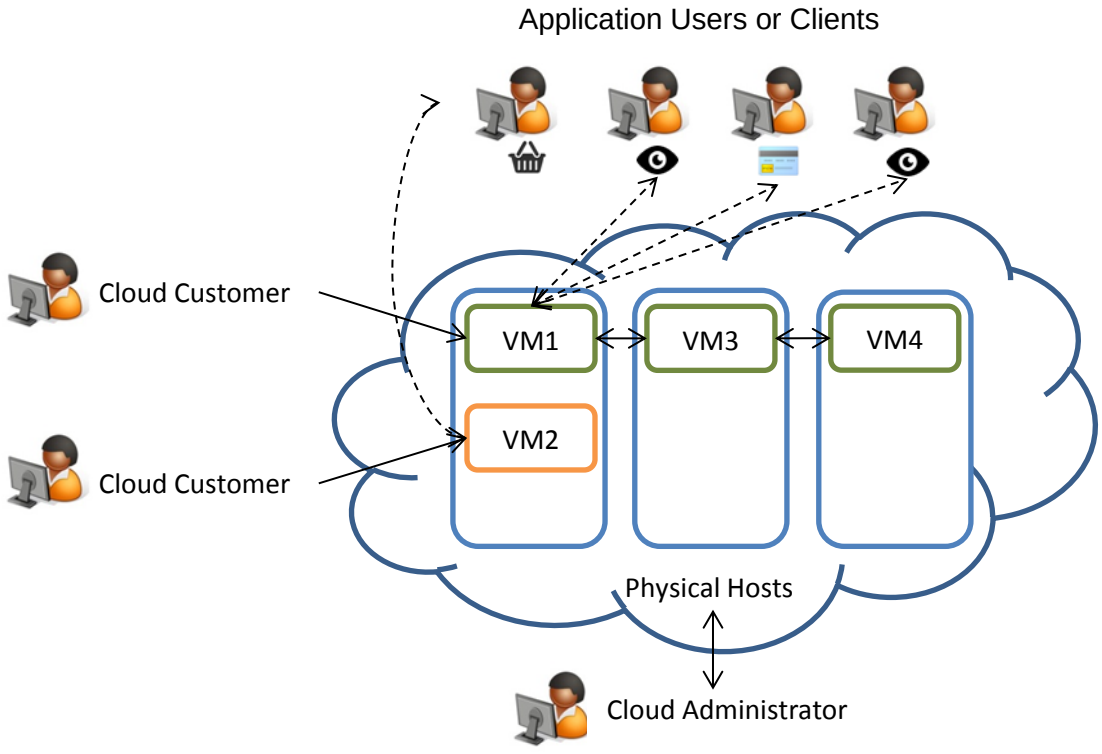


Figure 1-1: Simple overview of a cloud computing environment

In this thesis we place ourselves as the cloud administrator. The cloud administrator has access to the underlying datacenter hardware and has full control over where VMs are placed in the cloud environment. Cloud administrators have many responsibilities; for example, the reliability, availability, serviceability, and disaster recovery of the cloud environment. We focus on the performance of applications running in the cloud environment and the efficiency with which resources are utilized.

Resource utilization in cloud computing environments is maximized by placing multiple customers' VMs on a single physical host. For example, if two VMs each utilize 50% of a host's CPU they could be consolidated onto a single host. However, while high resource utilization lowers costs it also has a derogatory effect on applications' performances [7]. As customers share hosts they can degrade each other's application-level performances due to increased contention for shared resources. The contention of a shared resource is the amount of resource utilization not created by the customer's own VM. When resource contention is high, accessing resources will take longer and applications' processing times will increase. Figure 1-2 shows the degradation of an application's response time as the VM's host's CPU contention increases. As shown, the application's response time increases 400% from 300ms to 1200ms despite it having a constant workload level. Although administrators can minimize the cloud's resource waste by running every host at 100% resource utilization, the performance received by customers will be poor.

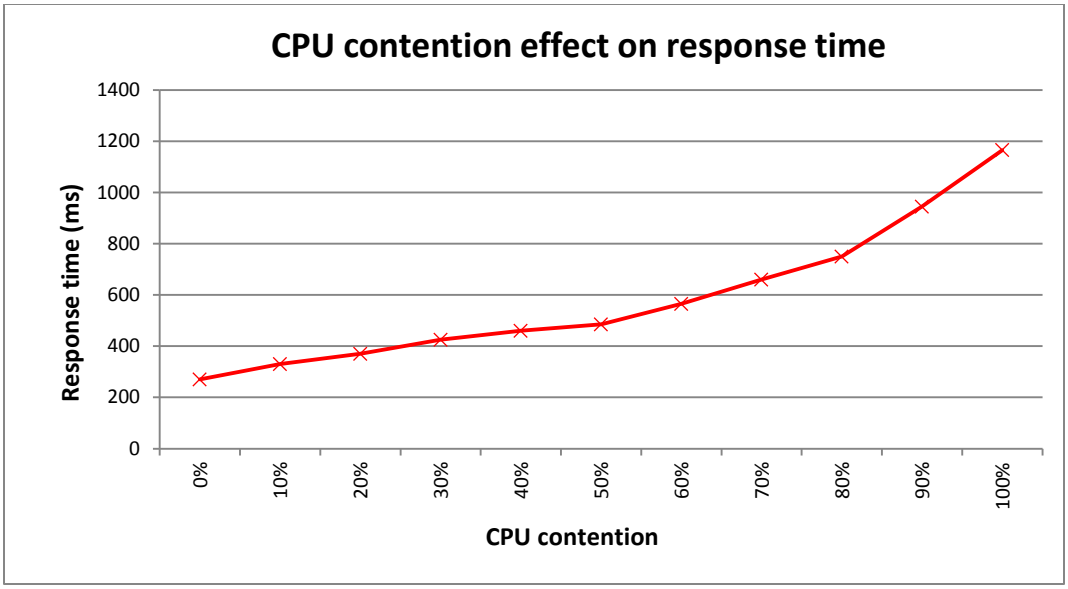


Figure 1-2: Effect of resource contention on application-level response time

An application frequently deployed in the cloud is a multi-tiered web application, shown in Figure 1-3. Splitting an application into multiple tiers allows separation for the application's data, data processing, and data presentation functions [8]. This layout is similar to the common Model-View-Controller layout used by programmers. It allows each part of the application to be edited or replaced without affecting other parts of the application. Splitting an application across multiple tiers and multiple VMs allows different tiers of an application to be sized differently depending on their individual resource requirements. For example, an application's logic tier may need five times the resources of its data tier. Cloud environments allow applications to be elastically scaled by adding or removing VMs at each tier of the application as needed [9]. This allows applications to quickly change their available resource levels depending on their workload level. In our work we assume that each VM runs a single tier of a single application as this reflects typical VM usage in real world cloud computing environments.

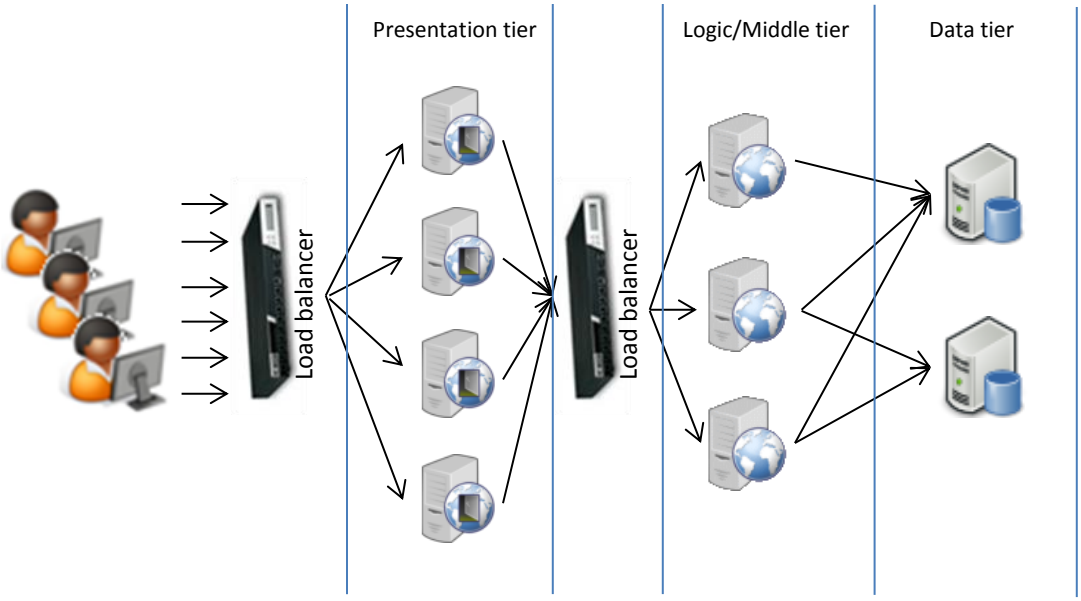


Figure 1-3: The layout of a multitier web application

1.1. Problem Statement

Although cloud computing environments allow customers to provision VMs on demand, the performance level of the VMs is not guaranteed. As applications' workloads vary over time the resources required by applications to maintain the same level of performance also varies. In addition, as customers share hosts' resources, increased resource utilization by one customer can cause performance degradations for others [7]. Due to these uncertainties, customers cannot currently ensure that their applications achieve satisfactory performance.

For web applications hosted in cloud environments, response time is the most common metric used to measure performance. Response time is the time taken for an application to process and respond to a user's request. Included in the total response time measurement is the processing time on the user's computer, the network round trip time, and the time to process

the request within the application itself. As a cloud environment cannot control the speed of a user's computer or the latency of the network, we refer to and measure response time as the time taken for an application to process a user's request. Other contributors to the response time are out of the cloud's control; therefore, out of the scope of our work and are not considered. This is the case for any cloud management scheme.

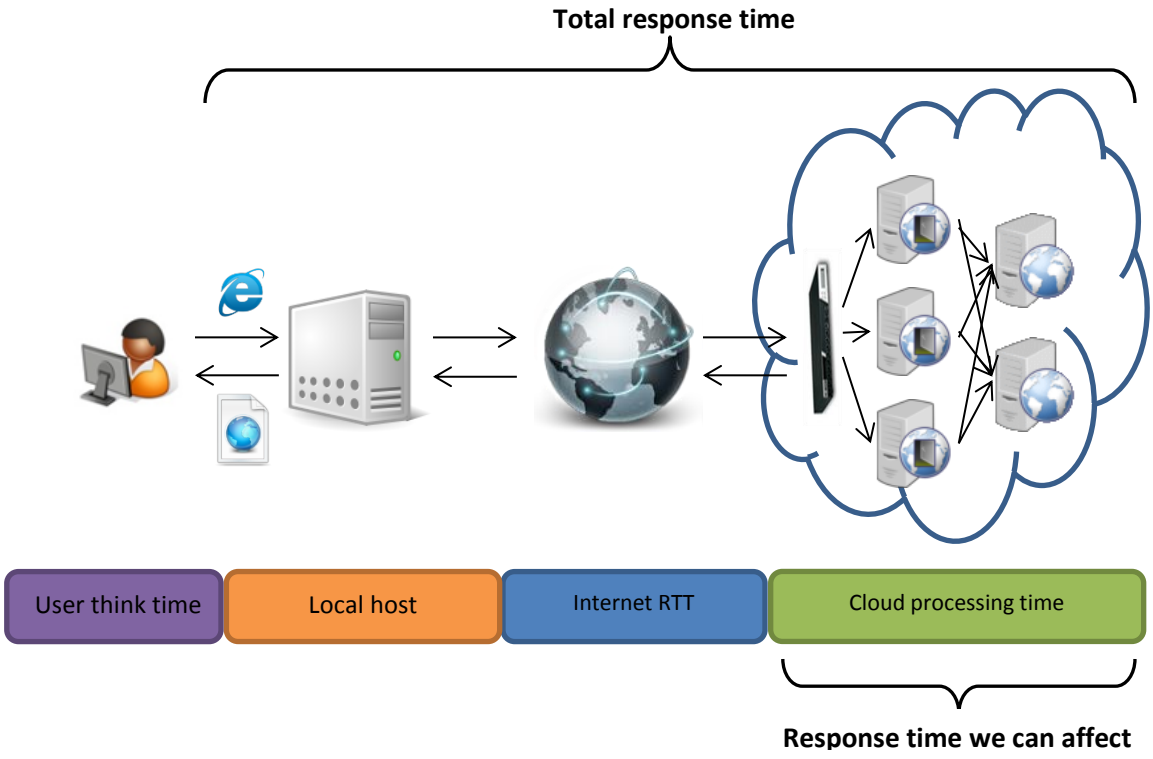


Figure 1-4: Total application response time and response time we can affect

Currently, cloud environments allow customers to specify their own level of available resources. Customers can request greater or fewer resources as they require them. However, it is more desirable to instead allow customers to specify the level of performance they wish to receive and request better or worse performance. This is because users perceive the

performance of an application through its response time, and are not concerned with the resource utilization of the cloud's hardware. Amazon has stated that it loses 1% of revenue per 100ms of additional response time delay [10]. To specify the performance goal of an application we use a Service Level Objective (SLO). An SLO is a target that an application should achieve to ensure that it is providing satisfactory quality of service to its users. An example SLO is that 90% of requests should be processed within 100ms during every 5 minute period. This SLO tells the cloud administrator the performance that the customer wants for their application, but does not specify the resources required to achieve it.

A simple method to achieve an SLO is to monitor an application's response time and incrementally increase its resource allocation until the SLO is achieved. However, this simple scheme would not work in real world environments due to a number of challenges:

- **Varying workload levels:** The number of users of applications varies over time. This changes the total workload of the application and will change the resources required to achieve its SLO.
- **Varying workload mix:** Applications typically serve many types of requests, each of which may consume different amounts of resources to process. If users begin submitting more computationally intensive requests the application will again require additional resources to achieve its SLO.
- **Varying workload arrival patterns:** Applications cannot control when users want to access them. If all users submit their requests instantaneously there will be a temporary spike in resource requirements.

- **Varying resource contention levels:** Even if an application has a stationary workload, variations in other applications' workloads that are sharing the same host can cause an application's performance to degrade.

In addition to the challenges stated above, there is also the challenge of using resources efficiently. Otherwise the cloud could provision every application with the maximum amount of resources it could ever need. However, this would be extremely wasteful, significantly more expensive, and the maximum resources required would still need to be calculated. As applications are typically multitier, every tier of an application must have its resources controlled to achieve an SLO. If we simply scale an application until the SLO is achieved we may be needlessly allocating resources to an application tier that does not need them. For example, in Figure 1-5 we can see that allocating more than 40% of a host's CPU to a web server has no additional improvement in its response time. This is because the web server is not the bottleneck of the application at this point, and resources are required at other tiers of the application to improve its performance.

In addition to addressing the challenges above, cloud management schemes must also be evaluated. There are already a number of web application benchmarks such as TPC-W [11], RUBiS [12], and Olio [13]. However, these benchmarks were designed to evaluate fixed resource installations, and not the elastic scaling environment seen in the cloud [14]. Additionally, the benchmarks were not designed using current cloud based technologies and design principles. Pugh et al. [15] conclude that "the amount of effort required to get RUBiS up and running outweighs the benchmark's usefulness at this point."

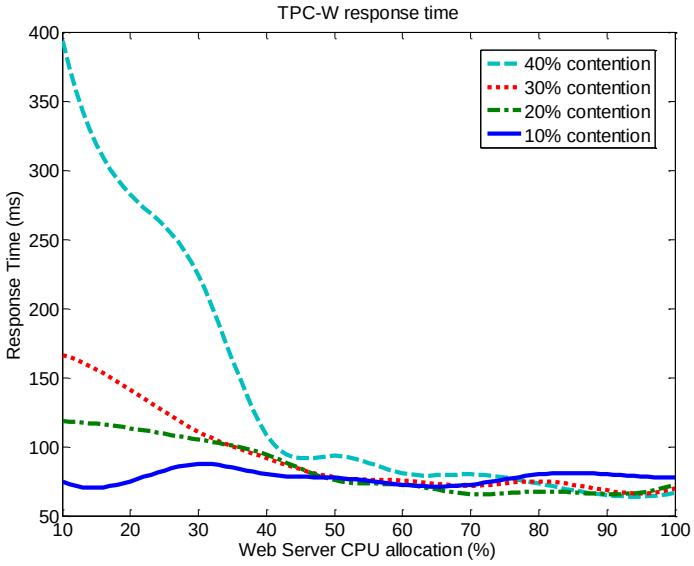


Figure 1-5: Allocating additional resources to an application does not always increase its performance

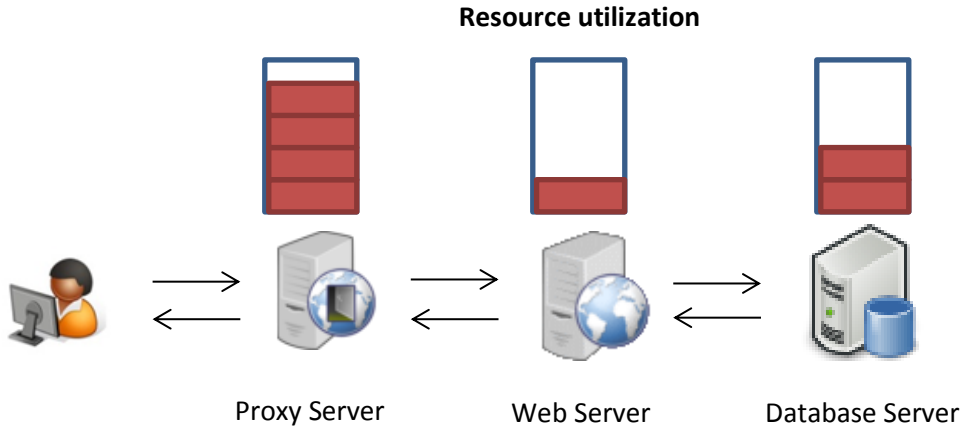


Figure 1-6: Resources must be allocated to the correct bottleneck tier

1.2. Solution Overview

To address the challenges discussed in the previous section and allow cloud computing environments to achieve customers' response time based SLOs, we propose a three pronged solution:

1. **Dynamic Resource Controller** that models applications' performances and reallocates the cloud's resources based on applications' current requirements. This allows applications to achieve constant performances despite variations in their workload levels, as well as variations in resource contention levels caused by other customers' workload variations.
2. **Input Shaper** that reduces the variability in VMs incoming workload levels and prevents hosts becoming overloaded. By filtering the types of requests being sent to a certain VM we can ensure that changes in workload mix do not overload a host's resources. By controlling the pattern of requests dispatched to VMs we can ensure that they do not cause significant resource contention and affect the SLOs of other customers' VMs.
3. **Carnegie Mellon Application Reference Test (C-MART)** is a new web application benchmark that is suitable to test cloud computing environments. It is designed to be elastically scalable to work in cloud computing environments. It utilizes modern technologies and design patterns to ensure its behavior closely mimics real world production web applications. It includes a sophisticated client

emulator that replicates realistic user behaviors to ensure the systems under test are evaluated thoroughly.

Figure 1-7 shows how the pieces of our solution are interconnected. First, we use C-MART to generate realistic traffic patterns. Emulated clients access the C-MART website and cause C-MART's VMs to consume a greater amount of resources. Before arriving at the C-MART website, each client's requests pass through the Input Shaper. This allows us to monitor the response time that each client receives. The Dynamic Resource Controller monitors each VM's and each host's resource utilization levels and collects the response time metrics from the Input Shaper. The Dynamic Resource Controller then decides if an application has sufficient resources to achieve its SLO. If an application does not have sufficient resources to achieve its SLO it is allocated more. If it is overachieving its SLO its resource allocation is reduced. The Dynamic Resource Controller then tells the Input Shaper to limit the number of requests sent to each VM. Excess requests are sent to an overflow zone where the application's SLO is not guaranteed. This reduces the variance in VMs' resource utilization levels to ensure that applications achieve their SLOs and that resource utilization remains high. The Input Shaper and Dynamic Resource Controller initially use the aggregate resources consumed by each application as a whole, but also work together to estimate the resources consumed by each incoming request type. This allows for a further reduction in resource utilization variance as the resources consumed by different request types can vary by an order of magnitude.

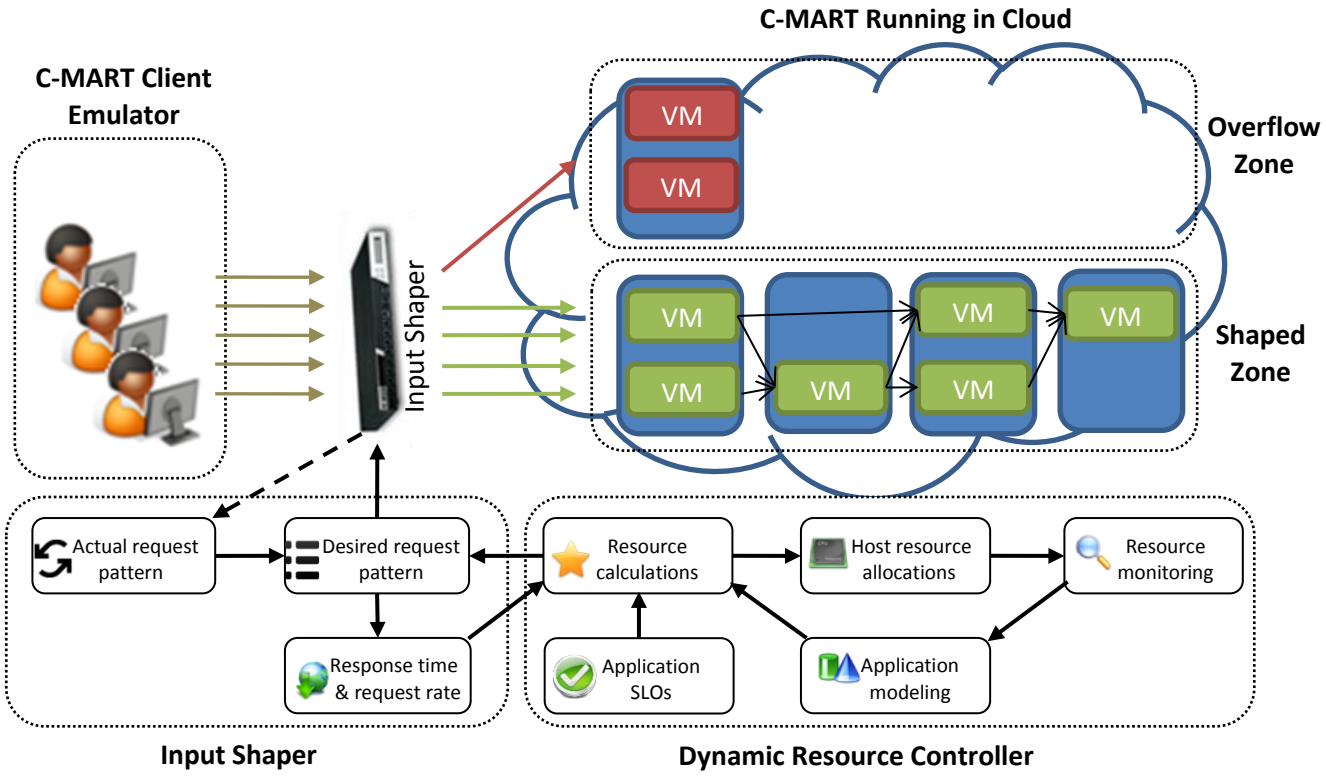


Figure 1-7: Solution overview

1.3. Contributions

1.3.1. C-MART

We propose C-MART, a new scalable multitier benchmark that is suitable for cloud computing environments. Although current application benchmarks exist, they were designed for traditional data centers where applications were hosted on dedicated hardware and were not elastically scaled. C-MART was created from the ground up using current technologies and designed specifically for cloud computing environments.

We validated C-MART by comparing it to the observed behavior of production and other benchmark systems. C-MART shows that validating cloud management systems using previous benchmarks could cause underprovisioning of resources 22% of the time and cause a 50% increase in performance estimation errors.

Additionally, C-MART models realistic user behaviors that are not present in other benchmarks. This allows systems to be tested under more realistic conditions, such as users leaving if performance is poor, or browsing the site through multiple tabs. This provides new testing scenarios metrics that can be used to evaluate the performance of cloud management systems.

1.3.2. Dynamic Resource Controller

We propose a dynamic resource controller for multitier applications running in cloud computing environments. Our controller models applications' response times and allocates them sufficient resources to achieve user-defined quality of service targets. This is in contrast to current cloud management systems that allocate a certain amount of resources to customers, but do not provide any performance guarantees.

We validate our controller by showing an application achieving a 100ms response time despite changes in its workload levels and varying amounts of resource contention from other customers' applications. We show that application response time can be improved by an average of 30% without consuming additional resources. Instead, the existing resources are allocated where needed to achieve satisfactory performance. We also show that applications' resource allocations can be on average 20% lower while still achieving quality of service targets.

1.3.3. Input Shaper

We propose a cloud Input Shaper that directs incoming workloads to two zones of the cloud datacenter. The Input Shaper shapes applications' incoming workloads to prevent hosts becoming overloaded and failing to achieve applications' quality of service targets. The shaper is an Apache HTTP server module that works with generic web applications running in cloud environments.

We validate our Input Shaper by showing that it can reduce resource utilization variance on hosts by 88%. This ensures that applications continue to achieve their SLOs despite changes in incoming workload levels. We also show that the Input Shaper can reduce resources wasted due to resource overprovisioning by 75%. Lastly we show that the Input Shaper can reduce the hardware required to achieve applications' SLOs by up to 45%.

1.4. Thesis Organization

This document is organized as follows. In Chapter 2, we discuss the background and related work relevant to this field. We provide an overview of cloud computing environments, currently proposed management schemes, and current application benchmarks. In Chapter 3, we present C-MART and demonstrate that previous benchmarks are not suitable for cloud computing environments. In Chapter 4, we present our Dynamic Resource Controller that allows applications to achieve SLOs despite changes in application workload levels and hosts' resource contention levels. In Chapter 5, we present the implementation of Input Shaper. The Input Shaper allows us to reduce the variability in hosts' resource utilization levels. This prevents hosts overloading, limits variability in resource contention, and ensures applications' SLO achievement. Lastly, we conclude in Chapter 6 with a discussion of future work.

2. Related Work

2.1. What is Cloud Computing?

The term 'cloud computing' is applied to many types of different computer systems. The fundamental idea behind cloud computing is to provide computing as an on-demand utility [6], in a similar way to how water and electricity are used. Clouds are typically built using commodity hardware to keep costs low. What differentiates a cloud service from a regular datacenter is the additional functionality that the clouds' software layer provides. Table 2-1 describes the three most popular supply models that current cloud providers offer [6].

Service model	**Description**
Software-as-a-Service (SaaS)	Provides users with access to fully functioning software, such as customer relationship management or e-mail software
Platform-as-a-Service (PaaS)	Provides users a platform upon which they can run their own programs, such as a web server or a database
Infrastructure-as-a-service (IaaS)	Provides users with computing resources which they can utilize however they choose, such as virtual machine running Linux OS

Table 2-1: Different types of cloud environments

The biggest advantage that cloud environments offer over traditional data centers is the ability to elastically scale applications. This allows customers to quickly scale up and scale down the amount of resources they have access to as their needs change. As customers only pay for

resources they consume, cloud computing can host customers' applications at low cost [6]. In this work, when we refer to cloud environments we are specifically referring to an Infrastructure-as-a-Service (IaaS) cloud. IaaS allows customers to run their own generic applications in the cloud and are extremely flexible. This type of cloud is the most popular and widely used, and is offered by companies such as Amazon [16], Microsoft [17] and Google [18].

2.2. Where Our Work Fits

Our work touches on a number of research areas related to cloud computing and datacenter networking. However, we believe that there is a gap in the current literature where our work fits. Most current research has focused on improving performance for one area of the datacenter, be it host resource utilization, network queuing delay, or minimizing VM migrations. Rather than focusing on improving performance for a particular piece of hardware, we attempt to improve the performance from the user's perspective. To achieve this, our work analyzes data from and controls both datacenter servers and network resources.

- Our **Resource Allocation Controller** differs as it attempts to achieve application-level SLOs and does not only attempt to maximize server resource utilization. It also considers the effect that contention for shared resources has on application-level performance.
- Our **Input Shaper** differs as it does not attempt to balance load, rather it sends the correct load based upon applications' current resource allocations. Also, the correct load is defined by application-level response time and not by host resource utilization. It also provides applications some isolation from resource

contention effects by dispatching traffic to different zones of the datacenter rather than simply admitting or dropping requests.

- **C-MART** differs as it is designed for cloud computing environments. It is scalable at each tier and reflects modern web application design. It is not designed to benchmark hardware, but instead evaluates the performance received by clients interacting with the system.

2.3. Dynamic Resource Allocation

There are many works on maximizing resource utilization levels and increasing efficiency in virtualized environments [19] [20] [21]. Existing commercial products are also available to facilitate the task of managing and relocating VMs. For example, VMware DRS [22] monitors the CPU and memory usage of VMs and migrates them to balance utilization levels. Similarly, VMware Distributed Power Management [23] minimizes the power usage of a data center by migrating VMs from under-utilized hosts and powers them off. Both of their systems focus on maintaining CPU and memory usage. Our work is concerned with service level performance, not just maximizing resource utilization regardless of its impact on performance. This section summarizes recent works related to the allocation of resources in cloud computing environments.

2.3.1. Placement Schemes

Placement schemes attempt to place applications' VMs such that the applications always achieve satisfactory performance. Gmach et al. [24] propose having multiple Quality-of-

Service (QoS) levels for applications. They attempt to achieve their resource-based QoS targets by placing VMs based on historic resource utilization levels. They do not consider dynamically allocating resources as demand varies. In further work [25], they migrate VMs based on their resource utilization levels. However, they only consider CPU bound VMs and do not consider application-level performance. Cherkasova et al. [26] place VMs based on their CPU utilization CDF. They note that by relaxing VM QoS requirements a small amount, the resources required to achieve the QoS target can be decreased by 25%. Tsakalozos's Nefeli [27] places VMs based on user defined 'hints' about applications' workloads, rather than solely based on their resource utilization levels. Williams et al. [28] perform an analysis of multiple VM placement algorithms. However, they only consider single tier applications with static resource requirements. Hyser et al. [29] note that VM placement cannot simply be a standalone optimization problem, as VMs already have an initial placement position. Entropy [30] considers VMs' initial placement positions and creates migration plans that contain the order with which VMs must be migrated.

2.3.2. Migration Based Schemes

Recent efforts such as [30], [31], [32] and [33] have attempted to increase resource utilization levels by migrating VMs. Each VM's resource utilization level is monitored and VMs are migrated to new hosts such that host resource utilization is maximized, and no host is overloaded. Kochut et al. [34] consider both autocorrelation and a periodogram to decide which VMs are the best candidates to be placed together. Ideally, co-located VMs should have a low probability of overloading the host. Hermenier et al. [30] consider the order that the migrations

occur, in addition to which VMs to migrate, to minimize the impact of migrations on system performance.

2.3.3. Overbooking Schemes

Another method to maximize hosts' resource utilization levels is by overbooking resources. Urgaonkar et al. [35] show that a 500% increase in utility can be achieved by overbooking hosts by 5% of their peak load values. This only causes a 4.6% decrease in overall throughput. However, the study focuses on a shared hosting environment, not a virtual one, and considers neither resource contention nor the overhead caused by a virtual environment.

Q-clouds [36] studies the effect of multiple QoS levels being available to applications. If resources are available, and a VM is willing to pay, its performance is improved by increasing its QoS level and assigning it more resources. Their results show that dynamic resource allocation can increase applications' average levels of performance. They conclude that VMs must have a balanced amount of resources; low enough to consolidate, but high enough to achieve satisfactory application-level performance. However, this ignores the ability to migrate VMs or redirect applications' incoming workloads.

2.3.4. Prediction Schemes

To maintain end-to-end service level performance, Stewart et al. [37] offer a response time prediction model. Their model is based on an identified trait model for multi-tier applications. Their work focuses on predicting the service response time based on pre-identified trait model relationships between processor properties and observed response time. Liu et al.

[38] use an autoregressive model to control CPU allocation. This allows VMs to be assigned a certain resource level to normalize multiple applications' performances. Padala et al. [39] and [40] have further used an autoregressive moving average to assign VMs multiple resources. They use applications' previous performance data to create a linear model that predicts application-level performance. Their work focuses on providing relative levels of performance between applications, especially in overload scenarios. For example, one application will always receive 50% of the performance level of another application, whatever that performance level might be. However, their work does not achieve application performance based SLOs. Our approach differs from these works as we probabilistically achieve response time based SLOs, rather than provide differentiated performances between applications. We also explicitly consider the effect of resource contention on applications' performances.

2.3.5. Achieving SLOs

Mylavarapu et al. [41] place VMs based on the probability that their combined resource utilization will not cause them to violate their SLOs due to a physical host's resources becoming exhausted. VMs are monitored to observe the stochastic nature of their workload levels. They are then placed such that there is a low probability that their combined resource utilization is greater than 100%. Although their work shows positive results, their SLOs only consider that combined resource utilization cannot exceed 100%. As resource utilization levels do not linearly map to application-level performance, it is unclear if their approach works for application-level SLOs. Watson et al. [42] calculate the probability that a VM has sufficient resources to achieve an SLO based on offline analysis of applications' historical performances. They assume that each

tier of an application consumes the same amount of resources. Chen et al. [43] note that SLOs can be decomposed and that each tier of an application can achieve a share of the performance target. They decompose applications' performances using a G/G/K queuing model. Our approach achieves SLOs through controlling applications' resource requirements, rather than only reacting to them. We control both server resource allocations and applications' incoming requests.

2.4. Input Shaping

Load balancing incoming requests is extremely common for large scale applications. As applications are typically distributed across multiple hosts their incoming requests must also be distributed across those hosts. Usually load balancing schemes attempt to distribute load as evenly as possible to ensure that each request receives an equal level of service. Examples of simple and commonly used load balancing schemes are Round Robin, Weighted Round Robin, and Least Loaded [44].

2.4.1. Admission Control

Kamra et al. [45] propose an admission control scheme based on a PI controller. The controller changes applications' incoming request levels to keep response times close to a set point. Their results show that dynamically controlling applications' numbers of incoming requests can significantly improve both response time and throughput. Mathur et al. [46] create an admission control scheme based on a feedback loop to maximize a power metric that balances application response time and throughput.

Huang et al. [47] perform admission control based on the expected service times of requests. Requests are placed in priority classes based upon their expected service times. Requests with the lowest expected service time are dispatched first to make expected delay proportional to expected service time. Gilly et al. [48] perform admission control based upon the expected resource utilization of servers. Requests are dispatched to a server only until its expected resource utilization reaches 100%. However, while this prevents overload, it does not ensure response time SLOs are achieved.

H. Zhang et al. [49] separate a video site's requests into two systems if the incoming number of requests are above the 95th-percentile of historic numbers of requests. Their request overflow system caches the top-k accessed videos to process as many requests with as few resources as possible. Peha et al. [50] also propose an admission control scheme for video traffic. Here the authors decide whether it is worth holding traffic in a queue to wait to be dispatched, or whether it should be dropped. The decision is made based upon the priority of the traffic and its expected time to transmit.

2.4.2. Load Balancing

Many load balancing schemes make decisions based upon the current requests present in its buffers. Yang et al. [51] filter incoming requests by type into classes, then prioritize based upon mean service time. This helps prevent small requests from being delayed behind requests that take a long time to process. Chi et al. [52] instead make their balancing decision based upon a cost function for each request class. This may mean that a larger request is dispatched before

a smaller request if the large request is more valuable. The value of each request is based upon a cost function of how much a customer is paying for their service.

Q. Zhang et al. [53] present a scheme to balance applications' incoming requests to minimize the number of correlated requests arriving at the same server. Their scheme does not consider the requests waiting in the load balancer's buffer, but instead considers the requests that servers are already processing. This improves applications' response times as it reduces the probability of multiple requests requiring simultaneous access to servers' resources. Shan et al. [54] also consider the requests that are already being processed by servers. They make their decision based upon the number of higher priority requests that a server is already processing. Requests are dispatched to the server where they will receive the lowest amount of queuing delay.

In [55] and [56] the authors reduce the variance in storage system utilization levels by decomposing incoming requests into two queues. A high-priority queue gets preferred access to the storage system, and a best effort queue that can use the system if there is spare capacity available. Our approach is somewhat similar to this on a datacenter level. In addition, we can control the amount of available resources rather than only react to what we are given.

2.5. Benchmark Applications

The purpose of a cloud benchmark is to help developers determine the right architecture, services, and settings for their applications by providing a testing platform which delivers relevant and comparable measurements [14]. This section summarizes existing

benchmarks and highlights some of the deficiencies that make them unsuitable for use as benchmarks for cloud management systems.

2.5.1. RUBiS

RUBiS [12] is an auction website modeled after eBay [57] that is used to evaluate the performance of various application design patterns. However, there are a number of flaws in RUBiS that make it unsuitable for use as a modern application benchmark [58]. For example, TCP connections are shared between multiple clients, which would not occur amongst real clients. Additionally, the think times for all page requests are determined by a single exponential distribution with a mean of seven seconds regardless of page content.

Cecchet et al. [59] analyzed the number of CSS, JavaScript, and Multimedia objects contained on the home page of RUBiS compared to the real application it emulates. RUBiS does not contain any CSS or JavaScript objects, and the number of multimedia objects is negligible compared to the real counterpart application. These objects significantly affect the response times of the website, therefore greatly reducing RUBiS's effectiveness.

RUBiS uses a closed-loop client generator, in which the number of concurrent emulated clients is constant in each experiment. This static workload pattern would not likely exist as client arrivals are not dependent on departures. The static behavior can produce overly optimistic results as knowing client data for one time interval unrealistically gives detailed knowledge for subsequent intervals. Additionally, the RUBiS data tier is configured as a non-scalable SQL database which is not representative of the distributed storage technologies used

in cloud computing environments. Pugh et al. [15] conclude that “the amount of effort required to get RUBiS up and running outweighs the benchmark’s usefulness at this point.”

2.5.2. TPC-W

TPC-W [11] is a transactional web benchmark that emulates an online bookstore. Its specification states, “TPC-W [represents] any industry that must market and sell a product or service over the internet… TPC-W does not attempt to be a model of how to build an actual application.” This design mantra results in TPC-W being a comparative benchmark between different sets of hardware. It is not designed to test application-level QoS in a cloud computing environment. It is noted in [58] that TPC-W’s workload generator shares a similar implementation to RUBiS’s workload generator and suffers from similar issues.

Binning et al. [14] concluded that the TPC benchmarks are not sufficient for analyzing novel cloud services. TPC-W does not represent modern Web applications, as it lacks a significant multimedia presence and client generated AJAX content. TPC-W compensates for this by disallowing caching, thereby increasing network traffic [11]. This ignores the crucial caching factors that affect real applications. TPC-W uses a SQL database and requires that ACID properties are enforced. Cloud computing systems do not always offer such strong transactional guarantees. TPC-W’s performance metric, Web Interactions Processed per Second (WIPS), is not of primary importance for cloud applications that are more concerned with scalability characteristics and QoS guarantees.

Figure 2-1 shows a comparison of the TPC-W homepage verses the current homepage of Amazon.com. It can easily be seen that TPC-W does not represent the state of modern websites.

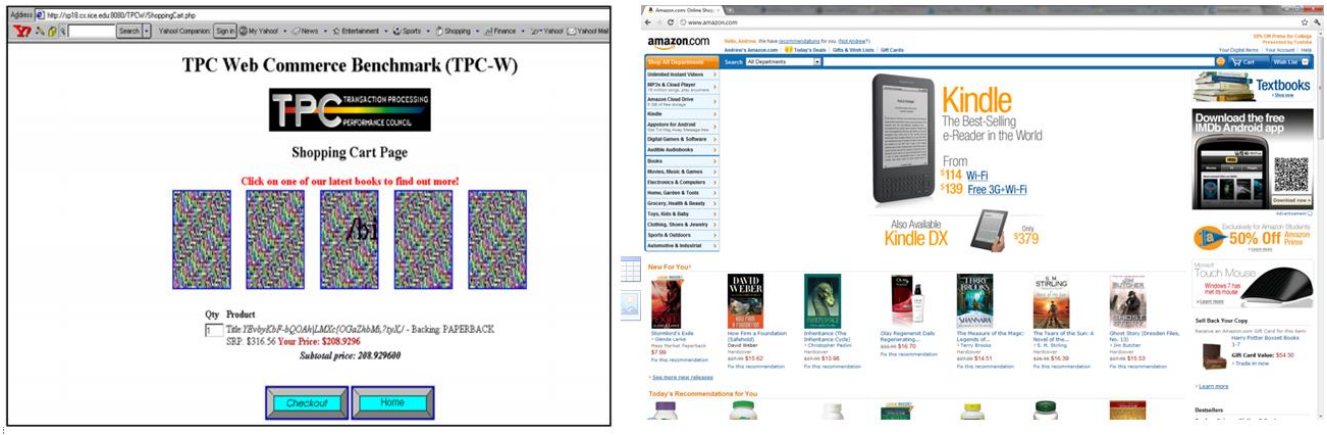


Figure 2-1: Screenshot comparison of TPC-W and Amazon

2.5.3. Cloudstone's Olio

Cloudstone's Olio [13] is an open-source social event calendar web application with Web 2.0 features. It utilizes some modern technologies such as AJAX and Memcache [60]. However, the application only has a limited number of functions, uses a single think time distribution, has a static probability matrix for page transitions, and does not capture or emulate SQLite. The database tier uses MySQL or PostgreSQL, and is scalable through read-only replicas, rather than the NoSQL approach that is increasingly popular in cloud environments [59]. A second workload generator, Rain [61], has been used with Olio. Rain allows the user to specify different workload mixes in different time intervals. However, there is no deviation from these as the generator does not consider the content from the page responses in determining future page transitions.

2.5.4. YCSB 2010

YCSB [62] is not an application emulation benchmark, but a framework for benchmarking different databases for cloud data storage. YCSB can be configured with various distributions of database operations (e.g. reads, inserts, deletes, etc.). It then benchmarks a database for throughput and response time performance. However, as a database-only benchmark it does not account for the interactions between the various tiers of applications. In production applications, it is difficult to predict the number of database transactions required per incoming client request. Additionally, this request number changes depending on clients' access patterns and the caching techniques used. Extrapolating application-level performance from only database performance is difficult, even if all other application tiers have overprovisioned resources.

2.5.5. SPECweb2009

SPECweb2009 [63] is designed to measure web server performance, specifically how it relates to power efficiency. It reports transactions as a function of power usage as its primary metric. It simulates a lightweight and efficient backend instead of using a traditional database and uses a closed-loop workload generator. They use a static Markov chain for page transition probabilities and all think times are based on a single exponential distribution.

2.5.6. CloudSim

CloudSim [64] is an extensible simulation toolkit that enables modeling and simulation of cloud computing environments. However, its application models are simplistic and do not

account for modern technologies. The simulated workloads do not produce the variability that would be observed in real world applications. There is also no SQL application model. As databases are an important part of any real application, this is a significant component missing in CloudSim which limits its testing abilities.

3. C-MART

Benchmark applications are essential for performance testing and validating systems before deployment to production environments. They emulate a typical production system and provide measurements of a secondary System Under Test (SUT) such as the processing power of a host, the maximum number of serviceable clients, or the efficiency of a resource allocation algorithm [6]. They allow various configurations, settings, and parameters to be compared so that the best values for a given scenario can be identified. Benchmark web applications typically consist of an example website, such as an online store or a social networking website, and a workload generator that sends traffic to the website in a manner emulating web-browsing clients. However, existing benchmarks were not designed for cloud computing environments [6] and therefore produce misleading results when benchmarking cloud management systems [14]. This can result in poor performance, resource underprovisioning, and Quality-of-Service (QoS) violations.

Benchmark applications are useful only if they accurately emulate the expected behavior of production environments. Existing benchmarks were designed for traditional dedicated hardware datacenters, not cloud computing environments. Their workload generators are simplistic and do not accurately represent user behaviors. The websites are therefore not excited with the variability of traffic patterns that would be observed in production environments. This produces overly optimistic results when used for systems testing. The benchmarks therefore fail to accurately validate the performance of a SUT.

In this chapter we present C-MART, a web application benchmark designed to evaluate systems running in cloud computing environments such as Infrastructure-as-a-Service or virtualized datacenters. We address four main concerns with existing benchmarks that make systems testing difficult, limited, and inaccurate. These concerns are *scalability, modern technologies*, *flexibility*, and *client realism*.

- **Scalability:** Cloud Computing environments allow on-demand resource provisioning [6]. As such, C-MART is designed to elastically scale at each tier of the application. It also includes a deployment server and scaling API to emulate a cloud computing environment in a local datacenter. Existing benchmarks are difficult or impossible to elastically scale.
- **Modern Technologies:** Current web applications utilize technologies such as AJAX, CSS, SQLite, and HTML5, and have multimedia-rich interfaces. Existing benchmarks do not utilize all of these features. This reduces the variability in their resource utilizations and response times, thus simplifying their management. We include these technologies so that C-MART emulates a modern application.
- **Flexibility:** C-MART is designed with multiple implementations of each tier. This allows a SUT to be validated under different architectures. C-MART can run as a two-tier system up to a six-tier system. Existing benchmarks have only a single design structure, thus limiting their testing scope.
- **Client Realism:** Clients accessing real world applications exhibit variable behaviors and access patterns. C-MART's client generator uses variable typing speeds, think times, browsing behaviors, QoS expectations, and page transition probabilities for each client.

The distributions used are derived from user studies and real world websites to ensure realistic client emulation. Previous benchmarks typically use only a single think time distribution for each page and a static Markov chain for determining page transition probabilities.

Feature	Current	C-MART	Use case	Impact
Client Caching	Low/ None	High/ SQLite	Load balance by request URL	**Current benchmark:** Low variability in response time 109 ± 138ms **C-MART:** High variability in response time 1720 ± 6100ms
Page access frequency	Static	Variable	Linear regression scheme to predict CPU Utilization based on request rate	**Current benchmark:** Regression scheme has 4.4% error **C-MART**: Regression scheme has 50% error Resources are underprovisioned since linear regression on request rate is insufficient for determining CPU utilization
Session based QoS	No	Yes	Determine profit based on clients completing browsing sessions	**Current benchmark:** Clients do not use QoS in decision making process and management is based on aggregation across clients **C-MART:** Individual QoS, causes client levels to change in open loop Can be used to relate resource provisioning to QoS
Page Content Variability	Low	High	Consolidating VMs based on average CPU utilization	**Current benchmark:** Violates SLA 0% of time **C-MART:** Violates SLA 22% of time Consolidation scheme would perform poorly in production system

Table 3-1: Overview of how C-MART features can identify problems in management systems missed by current benchmarks

Table 3-1 provides a summary of how C-MART's features allow it to identify problems in SUTs that are overlooked when using current benchmarks. We present examples where applications will be improperly allocated resources, causing either QoS violations or low resource utilization.

3.1. Scalability

A major benefit of cloud computing is the ability to provision resources on demand. This allows applications to quickly scale up and scale down as workloads vary. To emulate a cloud computing application, we ensure that C-MART can horizontally-scale at every application tier.

3.1.1. Tier Scalability

Figure 3-1 shows the architecture of C-MART when utilizing all of its possible tiers (flexibility is discussed in Section 3.3). Each tier of the application is scalable due to the design of the application and the underlying software, as shown in Table 3-2. The client can send requests

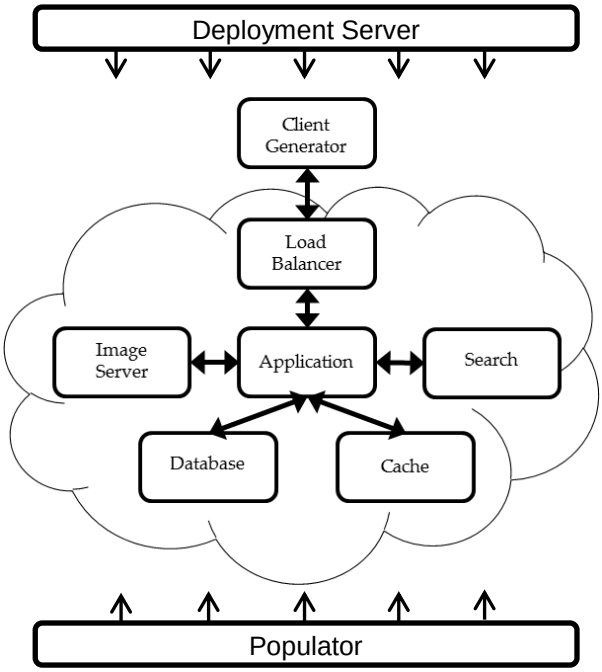


Figure 3-1: C-MART architecture. The client sends requests to the (up-to) six-tier application running in the Cloud

to different front-end load balancers using a system similar to round-robin DNS load balancing. The application tier is scalable as it is stateless, does not use user sessions, and does not acquire locks to shared resources on other tiers. All user state is stored at the data tier. This also allows the application tier to be quickly scaled down if the workload is reduced, as the tier retains no client data. The search, image, and cache tiers scale due to the underlying software, Solr [65], MongoDB [66], and Memcache [60] respectively. Lastly, the database tier scales as C-MART has a NoSQL implementation using Cassandra [67], again utilizing the scaling ability of the underlying software. Connection caching and limits can be configured via a web API.

Tier	**Description**
Client	Run on multiple hosts, consolidates statistics
Load Balancer	Clients can be directed to multiple load balancers, similar to DNS balancing system
App	Stateless, does not lock shared resources
Cache	Memcache Distributed Hash Table
Search	Solr Multiple read-only copies
Image	MongoDB replicas
Database	Cassandra replicas

Table 3-2: Scalability methods at each tier

3.1.2. Dynamic Scalability and Deployment

To reduce the complexity of scaling C-MART we have created a deployment and scaling API. Using preconfigured Virtual Machines (VMs) provided on the C-MART website [68] , the number of servers to be deployed at each tier can be defined using an XML description, as shown in Figure 3-2. The deployment descriptor can define hot and cold backup VMs, CPU and RAM allocations, and additional application specific information. During experiments, the scaling API can be used to add additional tiers to the application programmatically. This allows benchmark users to create custom datacenter management algorithms and dynamically provision resources. The APIs are provided for a KVM platform; however, we also provide installation scripts for use with any Linux operating system.

```
<Host hostOS="Fedora" port="4444" IP="10.66.1.3">
  <VirtualMachine IP="10.1.4.3">
    <OSType>Fedora</OSType>
    <VMType Type="Application" Backup="Hot"/>
    <UserName>root</UserName>
    <Password>cmart</Password>
  </VirtualMachine>
  <VirtualMachine IP="10.1.1.7">
    <OSType>Fedora</OSType>
    <VMType Type="SQL" RAMSIZE="512"/>
  </VirtualMachine>
</Host>
```

Figure 3-2: Example C-MART deployment file

The API simplifies the deployment and configuration of C-MART compared to other benchmarks. In C-MART, only the XML definition files need to be configured. C-MART tiers are then automatically deployed to each physical host, and configuration data, such as IP addresses

in the load balancer or my.cnf files in the MySQL servers, are automatically generated. This reduces the deployment time from hours with previous benchmarks, where IP addresses, network interfaces, and application-specific data need to be set manually, to only minutes required to define an XML document. Also, defining additional experiments is possible via the API, instead of editing the source code as required with RUBiS, for example.

3.2. Modern Technologies

To ensure that C-MART's results accurately represent those of a modern application, it is important to design it to accurately emulate a real world website using contemporary design methods and technologies.

3.2.1. HTML5, AJAX, CSS, Multimedia, and SQLite

Modern web applications utilize technologies such as HTML5, AJAX, CSS, rich multimedia, and SQLite. These technologies can have a significant impact on resource utilization levels and request access patterns, which ultimately affect clients' browsing experiences. For example, SQLite is used to locally cache dynamic data. This allows the remote application to only return data that has been updated since it was last locally cached. Without SQLite, the entire page must be rendered by the server for each request, consuming significantly more resources.

Multimedia presence on websites represents a large component of pages. A significant number of connections, round trip delays, network traffic, and disk Input/Output need to be consumed when the multimedia content is high, leading to a higher potential for QoS degradations.

HTML5 and AJAX automate or simplify processes that were previously created by a user-initiated request for an entire page to the server. HTML5 does advanced validation on forms reducing the number of pages regenerated due to input errors. AJAX requests can update specific page elements by requesting only certain information from the server, resulting in bursty database accesses depending on how many elements need to be updated and the time period since the client previously accessed the data. This prevents the entire page from being recreated for only partial page updates. AJAX requests may also be sent periodically, not initiated by a user page click. C-MART takes advantage of these design technologies. C-MART pages are formatted using CSS, JavaScript, and jQuery UI. On the Item page, prices are updated using periodic AJAX requests and item pictures scroll in a timed gallery. AJAX requests return JSON or XML objects from which JavaScript inserts the returned data into the existing page. Each C-MART page also provides statistics on how long different elements of the webpage took to generate. By comparison the RUBiS Item page is entirely HTML4 and always contains the same two images.

3.2.2. Real World Distributions

To allow clients to make content-based decisions that are representative of real-world clients, the data used to populate C-MART needs to emulate a production website. We sampled 100,000 eBay auctions to create empirical distributions of various website content including:

- Number and frequency of words in products' titles and descriptions
- Number of items in each product category
- Number of images on each product page

- Product bid values and buy now prices
- Seller ratings

These distributions cause data hot-spots to naturally form within the website as emulated clients use the page content and item data to differentiate between items and rate their appeal. The RUBiS item titles and descriptions are instead generic and repeated across items making them essentially identical.

3.3. Flexibility

Current benchmarks are designed with a single architecture and typically allow only minor configuration details to be altered. C-MART is instead designed with a highly customizable architecture, determined by simply setting flags in its configuration file. This allows a relative comparison of how different application architectures and technologies will perform with the SUT.

3.3.1. Tier Configuration

Not all application tiers are required when running C-MART. It can function as a two-tier Application and Database server configuration, up to the six-tier architecture shown in Figure 3-1. In addition, some tiers have multiple options for the underlying technology as C-MART has been implemented using multiple architectures. For example, the database tier has both a SQL and NoSQL implementation that can be chosen using a single flag. This allows C-MART to emulate both a traditional application using a relational data store and a modern application using a cloud storage engine. Any additional storage engine could be used by implementing a

provided abstract class. Populators are provided to prepopulate the databases with a configurable amount of data before each experimental run. Table 3-3 provides an overview of the architecture options that can be configured for the different implementations of C-MART.

Flag	Effect
Solr	On/Off to use Solr as the site's search engine
Cache	0, 1, 2 use Memcache to cache either none, database heavy query results, or all results
Database	MyISAM, InnoDB for MySQL storage, Cassandra for NoSQL
Image	'img' for local storage, 'netimg' for NFS, 'mongoDB' for GridFS/MongoDB
Web	Web 1.0 or Web 2.0 enable HTML5, SQLite, AJAX, JavaScript

Table 3-3: A sample of C-MART's configuration flags for different C-MART implementations

3.3.2. Web Design Technologies

In order to analyze the effects of different web design technologies, C-MART includes two web implementations with identical functionality but using different design architectures. The first implementation is a traditional client-server model website where each page request is entirely processed and rendered at the server. The second implementation renders the pages locally, more heavily uses JavaScript, CSS, and AJAX, and increasingly relies on SQLite, as described in Section 3.2.1.

3.3.3. Client Flexibility

The workload generator has a number of different operating modes to test different client behaviors and access patterns. The workload generator can be run in an open-loop or

closed-loop mode with either a static or time varied number of clients. Users can also enable an option to create random bursts of clients, known as flash crowds or 'the Slashdot effect' [69]. Clients can operate as individuals with complex decision making processes (see Section 3.4) or use a simple Markov chain to determine page transitions. There are also preconfigured read-heavy and write-heavy biases that can be enabled in conjunction with any of the other options. Users can define client satisfaction levels that allow clients to behave differently if they receive poor response times from the website.

3.3.4. Experiment Repeatability

C-MART experiments are logged for repeatability so management algorithms can be directly compared. Clients' actions and think times from experimental runs are output to XML files which can be read by the workload generator and reproduced in subsequent runs. Bid values and exact items chosen may be modified on the repeated runs to avoid concurrency problems on the server.

3.4. Client Realism

Real clients exhibit unique behaviors and have different expectations of performance when interacting with a website. The variability of clients creates variability in the resource utilization of the application and ultimately changes how applications need to be managed. We highlight factors that influence client behaviors and their effects on server utilizations and management systems.

3.4.1. Content and History Based User Decisions

The content of a specific page and to an extent the entire website influences client behavior. For example, an item with a good review, thorough description, and multiple high resolution photographs is a more attractive item than one with a short description sold by a user with negative reviews. A client is more likely to bid on the former over the latter. Also, a client's prior actions influence their current decision making process. For example, a client is more likely to bid on an item they have been outbid on than another random item. It is therefore inappropriate to model page transition probabilities as a simple Markov model according to page type as current benchmark workload generators do. Clients in C-MART analyze page content and track their previous decisions when determining page transition probabilities. Knowing only a client's current page request type is therefore insufficient knowledge to use in predicting their next action. Some content-based decisions can create bursts in traffic and data hot-spots. Clients are much more likely to bid on an item as auction time nears expiry, creating bidding wars which produce a large traffic burst to a single page.

As stated previously, empirical distributions are used from eBay to create the content for each item. The clients are set to make decisions by slightly different criteria. For example, some clients may only buy from sellers with an extremely high rating while others consider the quality of an item's description to be of the greatest importance. Clients also have different pricing criteria. Two clients may find the same item equally enticing but only one may bid as the item may be out of the other's price range. As different webpages cause different amounts of

server resource utilization, the variable client behavior increases the difficulty of managing the application.

3.4.2. QoS-based User Decisions

Amazon loses 1% of revenue for every 100 ms of additional delay on response time [10]. When clients view more items and pages load faster, they are more likely to find and buy a desired product. Conversely, clients leave slow websites. Keeping clients active also generates ad revenue with increased page clicks. Clients' traffic patterns therefore change depending on the response time. The faster a website responds, the more pages users will attempt to access. For example, a user may compare a different number of related items, which changes the amount of information a client has access to. This ultimately weighs on the final bidding/purchasing decision.

Each C-MART client has their own expectations of the website's performance; some are more patient than others. Clients also have different performance expectations for different pages. Clients know that processing credit card information typically takes longer than bringing up an item page and are therefore more patient. These behaviors are particularly important when using the open-loop client generator. In the open-loop generator, client arrivals and departures are independent of each other. Therefore, QoS-related decisions are extremely important to maintain high client levels. Current benchmarks use closed-loop client generators where a client departing is instantly replaced by another arriving. This fails to account for QoS effects as poor website service does not cause a decrease in the number of active clients. A

management scheme would then ignore that many clients leave due to dissatisfaction with the website experience.

3.4.3. Modern Browsers

Modern Internet browsers support tabbed browsing which can change clients' behaviors. Clients may open many pages at a time, for instance from a search page to compare multiple items. This causes large bursts in traffic followed by a long idle period as the client examines each opened tab. It also means the search page does not need to be reloaded between items. Browsers also allow clients to automatically fill out online forms. Typing speed and typing error rate are two factors used to create variable clients. However, autocomplete form fillers allow clients to enter data extremely quickly. This creates multimodal think time distributions on form pages which make for highly variable access patterns.

Clients initially accessing a website do not necessarily arrive with an empty data slate. Clients may have previously accessed the site and would therefore already have a prepopulated cache when they arrive. This is extremely important for the initial access when most of the JavaScript, CSS, and common image files are initially downloaded. The number of clients who arrive with prepopulated caches is a tunable parameter in C-MART.

3.5. Performance Metrics

It is important to use proper metrics when evaluating the performance of a benchmark. Metrics such as WIPS which may have been useful for benchmarking hardware are inadequate for evaluating the performance of an application in the cloud.

We provide a number of different metrics to evaluate the benchmark performance. We provide distributions of the resulting response times for each page. For each client we also provide the length of the browsing session, how many pages were visited, and whether or not the client left due to dissatisfaction with the QoS. We report the server utilization levels for each tier and the distribution of how long it took to build each component of the pages, for example the times to process the parameters, access the database, process the page data, and render the page.

3.6. Implementation

C-MART is implemented using widely used enterprise software to ensure it mimics a real world production system as accurately as possible. We chose technologies such as Apache HTTP Server [70], Apache Tomcat [71], and Apache Memcache [72] as they are prevalent in the business environment. As of 2013 it is estimated that Apache HTTP server is used by over 50% of websites worldwide [73]. Both C-MART's client and server code is written in Java as it is another technology prevalent in business applications. The client side implementation uses jQuery which is estimated to be used in 65% of the top 10,000 websites [74].

Figure 3-3 shows a screen shot of C-MART next to a screen shot of RUBiS. It can be seen that C-MART's implemented site features and content are much closer to modern websites' designs than RUBiS.

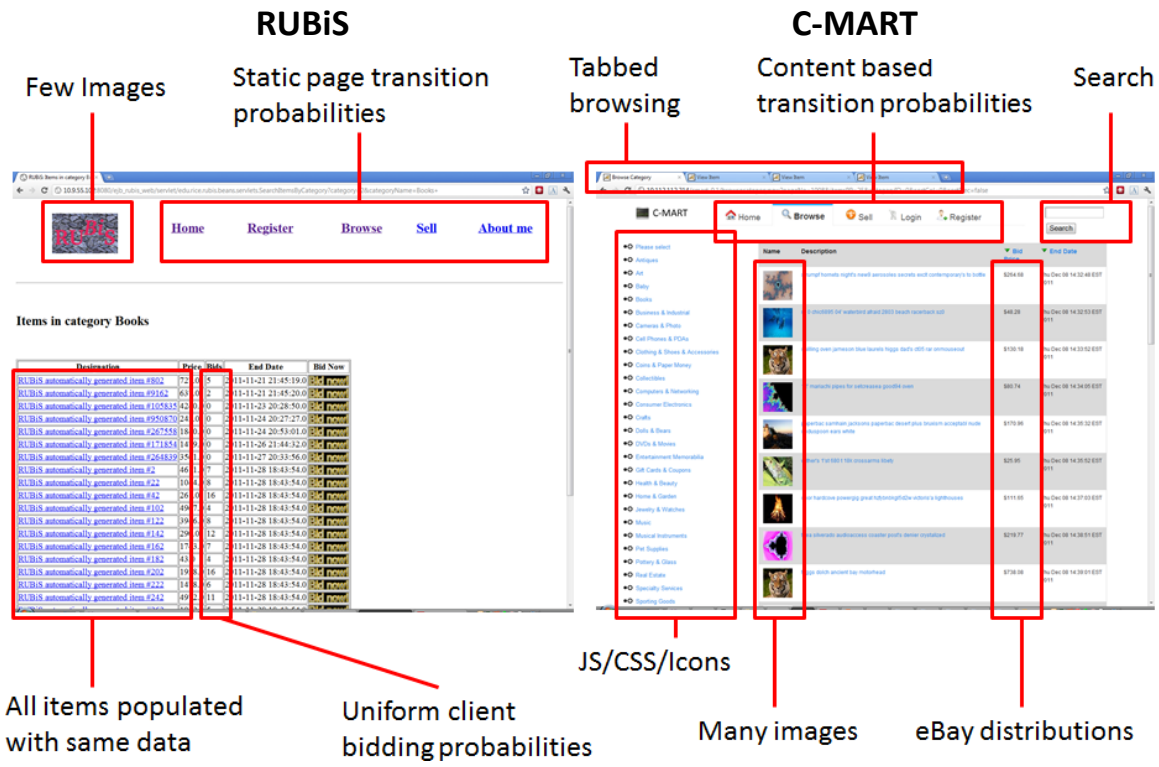


Figure 3-3: C-MART screen shot

The C-MART client emulator is a Java JAR file that can be run from the command line, allowing clients to be easily started by users across multiple hosts. The client emulator reads configuration values from a text file and creates a data output directory for each test execution. Figure 3-4 shows the basic components of the client emulator. As the client is highly configurable, not all components are used for every test. Components connected with dashed arrows are optional and can be enabled and disabled to further alter the client's browsing behaviors. The different colored data processing elements indicate the various different client configurations that can be running simultaneously. This allows systems to be tested with a mixture of client types ensuring the application is performing well for various types of users.

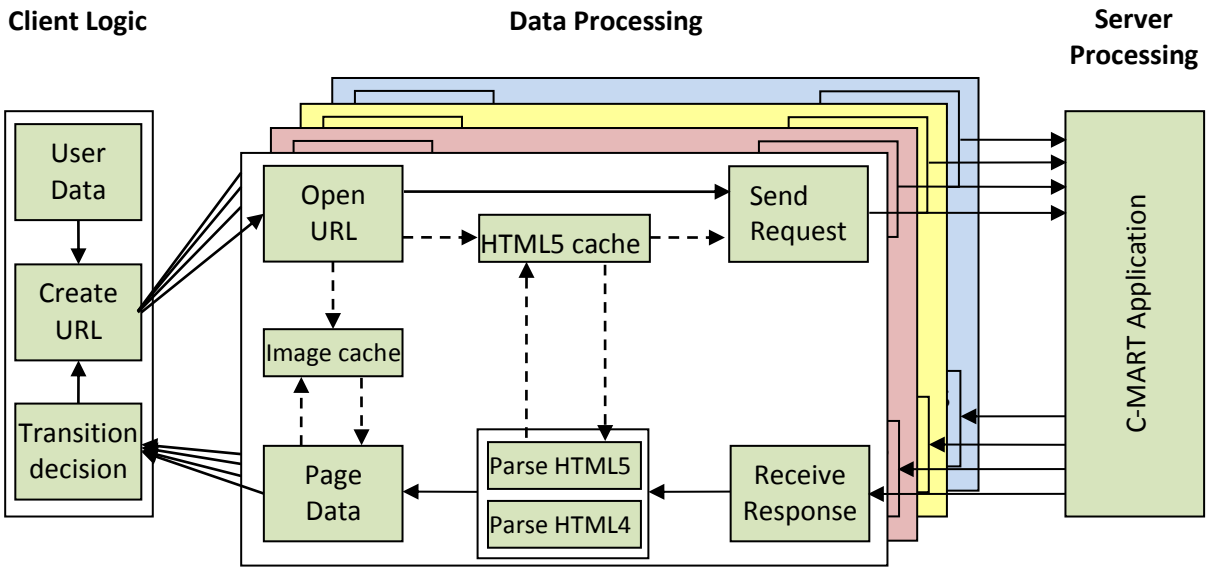


Figure 3-4: Client implementation overview

When a client is first created, it always accesses C-MART's home page. From there it makes a page transition decision based on its configuration profile. Clients will access the site to browse, buy, sell, or a mixture of each. After the client has made its initial page transition decision, it makes future decisions based on page content rather than a predetermined transition matrix. For example, if a client has previously bid on an item, the bid will be listed in their account history. A client is likely to check the status of their previous bids when they login to view and potentially re-bid on them. If a client has multiple items in their bidding history, they are likely to visit each of them.

The data processing element of the client emulator retrieves and processes the webpage data. When it receives a URL to open, it first checks the cache to see if the client already has the data being requested. This is similar to a browser's cache. If the data is present it is returned from the cache rather than accessing the website. For HTML5 requests, the contents

of the cache can change the type and content of requests sent to the server. For example, if the client's bidding history is being updated, then only bids that have occurred since the last update are required. This causes HTML5 requests to exhibit significantly different behaviors than HTML4 requests. Whereas HTML4 requests will reload all of a page's dynamic data each time the page is accessed, HTML5 will load all of the data only the first time a page is accessed and will subsequently only update the data. This is why the client emulator has two different implementations of internal data processors and caches.

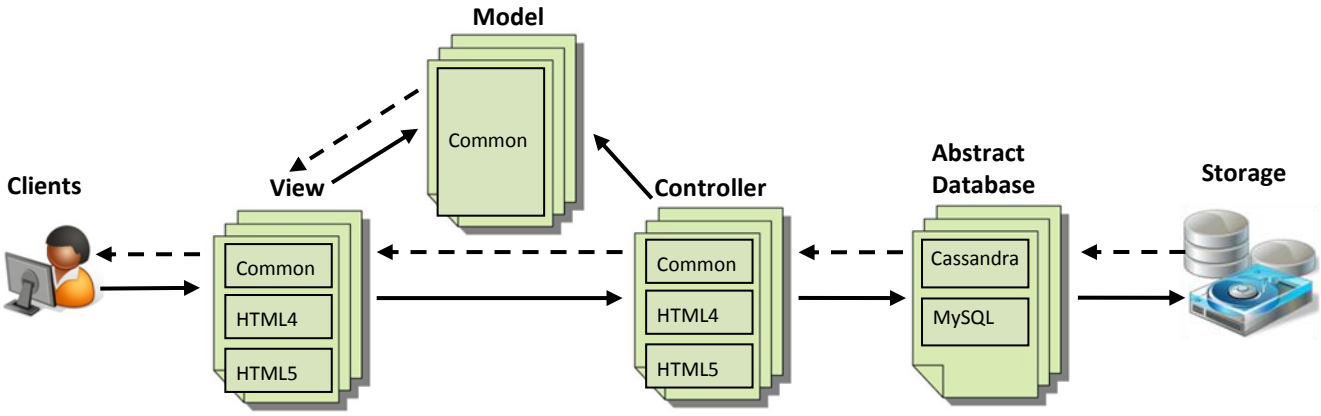


Figure 3-5: Overview of C-MART's server implementation

The server side implementation is a multitier application that follows the Model-View-Controller (MVC) architecture. The MVC architecture is one of the most commonly used software architectures in real world production applications. Using this architecture ensures C-MART's design is representative of real world applications. In addition, it also helps C-MART's code base remain manageable and flexible despite its large size. The MVC architecture allows the HTML4 and HTML5 versions of C-MART's server application to share large amounts of

common code. This ensures that the differences in observed behaviors are due to the features present in the underlying technologies and not in the individual implementations of C-MART.

C-MART's application logic is implemented using Servlets in Java 6. Java 6 Servlets were chosen as they are commonly used in business applications and are supported by cloud computing environments such as Google's App Engine. The Servlets are implemented in a stateless fashion; data is either stored on the data tier or discarded between requests. This is consistent with App Engine's default instances, where multiple requests from a single client are not guaranteed to be dispatched to the same application instance. Applications are increasingly being designed with stateless application tiers as it allows resource allocation decisions to take effect more quickly. Application tier instances (or VMs) can be create or destroyed on demand without the risk of losing clients' data. In addition, the number of requests dispatched to each application tier instance can be dynamically changed without having to migrate clients' data.

C-MART's runtime configuration can be reconfigured via the URLs passed by the client generator. This allows the version of HTML being used to be changed via a single variable. The code that parses the request parameters passed by the clients is common to both the HTML4 and HTML5 implementations. After the parameters are parsed, each page performs different logic depending on whether the HTML4 or HTML5 implementation of the page was specified.

Cloud computing environments frequently use non-relational databases for data storage as they are easy to scale-out. C-MART's design allows the database used by the data tier to be changed by a single variable. C-MART can therefore use either a relational or non-relational database with either version of HTML4 or HTML5. To achieve this, all of C-MART's database

operations are performed through an abstract database connector class. Any data storage system can be used as the data tier, provided that it implements the abstract database connector class. We have currently implemented two database connectors: a MySQL and a Cassandra connector.

To ensure the results reported by C-MART are accurate and are not simply random artifacts we need to ensure that C-MART's code is as bug free as possible. To help achieve this we created roughly 10,000 lines of test code across twenty-three different test classes. C-MART contains more test code than the entire code base of RUBiS. Each individual page in C-MART is tested for both its HTML4 and HTML5 functionality. The abstract database class is also tested ensuring both the MySQL and Cassandra versions of C-MART operate correctly. In addition to unit and functional testing we also performed performance testing to ensure each different implementation of C-MART has comparable performance. While some performance differences can be expected due to the different underlying technologies, each implementation performs representatively to its underlying technologies. We also tested C-MART with over 100 manual user tests to ensure the user interface returns the results that a user would expect. In addition to formal testing, we have run C-MART thousands of times during development and testing and are confident in its reported results.

3.7. Experimental Results

To identify where current benchmarks may produce inaccurate results for existing management schemes, we run C-MART against current benchmarks for a number of common datacenter scenarios. By comparing the results, we identify where the SUT fails to react

correctly in a production environment when it is tested using current benchmarks rather than C-MART. We begin by outlining a number of common management techniques and algorithms and how they are applied to cloud applications.

We use a custom data center with a flat local area network using commodity hardware. The physical hosts' OS is Red Hat 6.32 and runs the KVM hypervisor. Each server has a dual-core 3.2 GHz CPU with 4 GB of RAM. The hosts are connected via a Gigabit switch. Each VM has one virtual CPU and is allocated 1 GB of memory.

3.7.1. Management Systems

Cloud management systems involve procedures for VM placement, VM migration, resource provisioning, cost-optimization, energy-optimization, and achieving SLAs and QoS targets. We detail the techniques and algorithms used by these management systems and how their performance is changed when tested with C-MART instead of the existing benchmarks.

Controllers have been developed to model application response time as functions of resource utilizations (CPU, memory, I/O, network). These functions are used to adjust applications' resource allocation levels to drive response times to target values [40] [75]. These models typically assume that response time is a deterministic function of resource utilizations, which we show is not a valid assumption for dynamic web pages. Additionally, response time is often modeled using the total resource utilization during a time interval. This method is only suitable if the ratios of requests for each page type are static over time. However, real websites have highly varied and non-stationary page access patterns that make these deterministic models impractical to use [76].

Some management systems model QoS metrics as a function of request type. Works such as [77] model the response time of each web page as a linear function of the page request rate. This is a reasonable assumption when each request consumes the same amount of resources as is common in existing benchmarks. However, due to effects from multimedia, caching, and SQLite, the amount of resources consumed for two identical page requests at different times or from different clients can be drastically different. Therefore, calculating response time as a deterministic function of request rate does not produce accurate results when used with a real application.

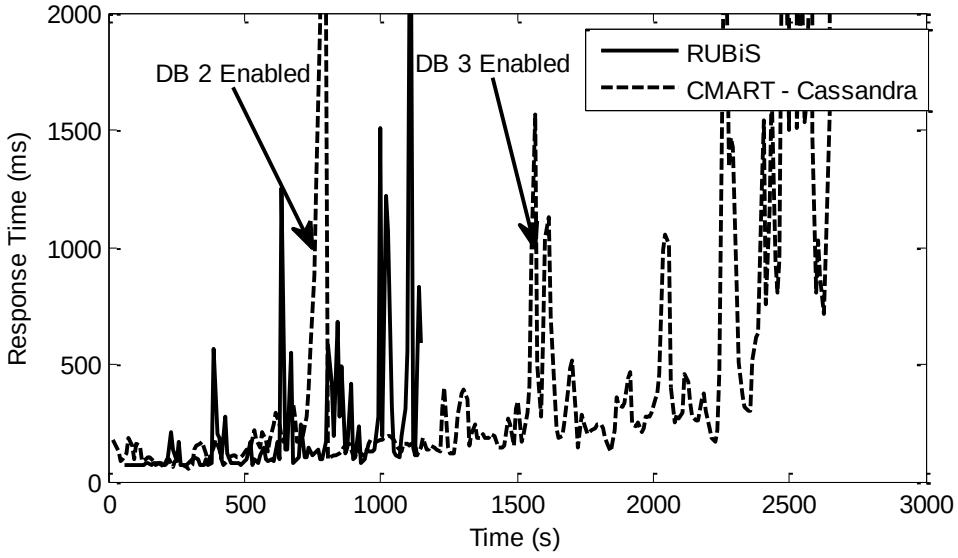


Figure 3-6: C-MART scalability. Additional database instances are activated when response time degrades

Autocorrelation of response time and resource utilizations [34] is used in management systems to predict future performance. This is effective when clients have predictable access patterns, as is the case with benchmarks that use static page transition probability distributions. However, it is much less effective when clients' behaviors are less predictable. Larger variance in

the resource consumption of each page also makes the use of autocorrelation techniques more difficult. It is therefore important to test such management systems with a realistic benchmark to fully validate their potential performances.

3.7.2. Application Scaling

A major benefit of cloud computing environments is the ability to elastically scale applications to react to changes in workload levels. A benchmark application must be able to scale to utilize a large number of servers. C-MART is designed to horizontally scale at every tier. This prevents any one tier from becoming a performance bottleneck. A major component of this is the ability to run either SQL or NoSQL database storage. Current web benchmarks utilize SQL databases that can be difficult to scale. NoSQL storage allows users to dynamically add resources to their storage tier, redistributing storage keys and data replicas to evenly redistribute load. NoSQL is commonly found in cloud computing environments such as Google's Big Table or Amazon's Dynamo.

Figure 3-6 shows the 95th-percentile response time of both C-MART and RUBiS as their workloads are increased. RUBiS is configured with one load balancer, six App VMs, and one database VM. C-MART is configured with one load balancer, one Solr VM, six App VMs, and three Cassandra VMs. RUBiS only receives one VM on its data tier as this is its default configuration. We modified the RUBiS client generator to linearly increase the number of emulated clients over time. We run C-MART initially with only one Cassandra VM. As application response time increases we enable additional Cassandra VMs. As expected, having the flexibility to add additional processing and storage power to the data tier allows C-MART to scale to three

times as many database instances. In addition, C-MART sends on average 8 kB of data per request compared to only 0.8 kB for RUBiS.

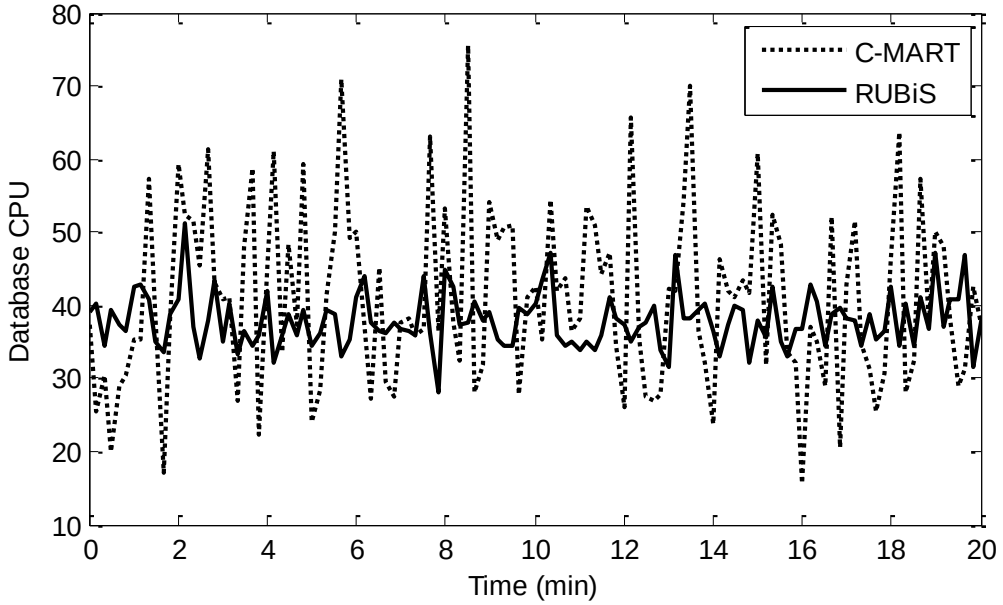


Figure 3-8: C-MART and RUBiS database CPU for a static client level at same CPU average

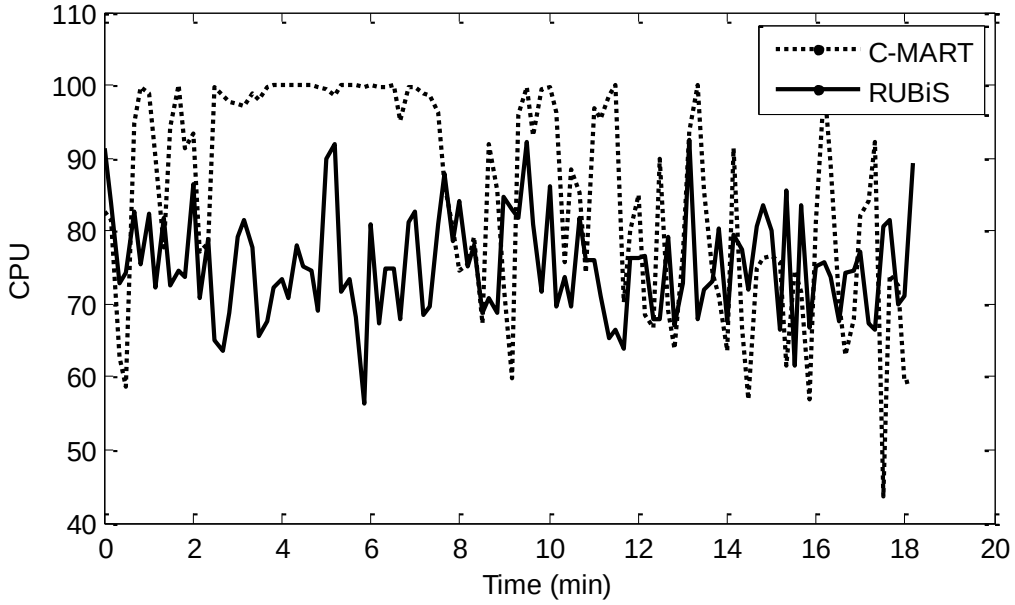


Figure 3-7: CPU of two consolidated instances of C-MART and RUBiS

3.7.3. VM Consolidation

Cloud computing environments rely on VM consolidation to achieve high resource utilization. This consolidation reduces the amount of hardware required to run a set of applications and reduces energy consumption. To satisfactorily consolidate VMs, one must ensure that all VMs receive a sufficient amount of a host's resources. Otherwise applications' performances will degrade resulting in QoS violations and unsatisfied clients.

Figure 3-8 shows the CPU utilization for both C-MART and RUBiS for a static number of clients over a twenty minute time interval. The average CPU is equal for both benchmarks at 40%; however, the standard deviation is 340% greater with C-MART. This is due to the complex client request patterns and use of technologies such as AJAX and SQLite. This experiment was repeated with Olio where the C-MART CPU standard deviation was similarly 286% greater. This increased variability makes VM consolidation difficult. Figure 3-7 shows the result of what happens when two C-MART or RUBiS VMs are co-located on a server. All VMs were run in the same mode as for Figure 3-8. It can be seen that the RUBiS VMs consolidate satisfactorily as there is always CPU unutilized. However, because of the increased variance, the CPU is exhausted in 22% of the measurement intervals when consolidating the C-MART instances. This results in poor response times for clients. This is also evident from the 90^{th} percentile response times shown in Figure 3-9 for the single VM experiment, where the standard deviation of response time is five times greater for C-MART than for RUBiS. These results illustrate the resource under-provisioning that may occur if the SUT is not properly validated. Using a current

benchmark, average CPU utilization appears to be a sufficient metric for VM consolidation; however, C-MART identifies that this is not true.

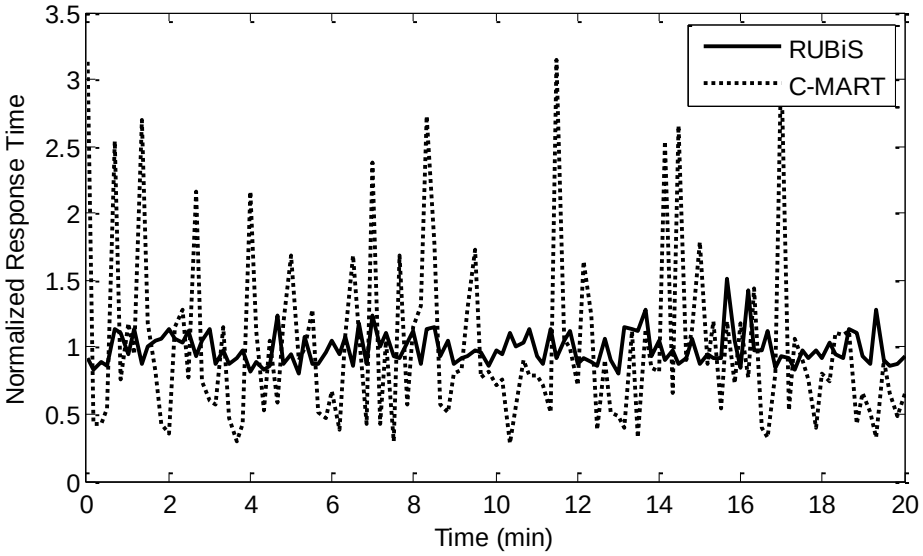


Figure 3-9: C-MART and RUBiS response times for a static client level
For comparison purposes, each response time was normalized to the average

Figure 3-10 shows the significant impact that multimedia content can have on webpage response time. Here C-MART is run with its images, CSS and JavaScript first enabled for download and then disabled. It can be seen that the application's response time is not only greater due to the additional multimedia content, but the variance of the response time is also significantly increased. Testing systems with benchmarks that do not contain significant multimedia content will produce overly optimistic results. This results in poor estimates of an application's resource requirements in production environments causing resource under-provisioning and QoS violations.

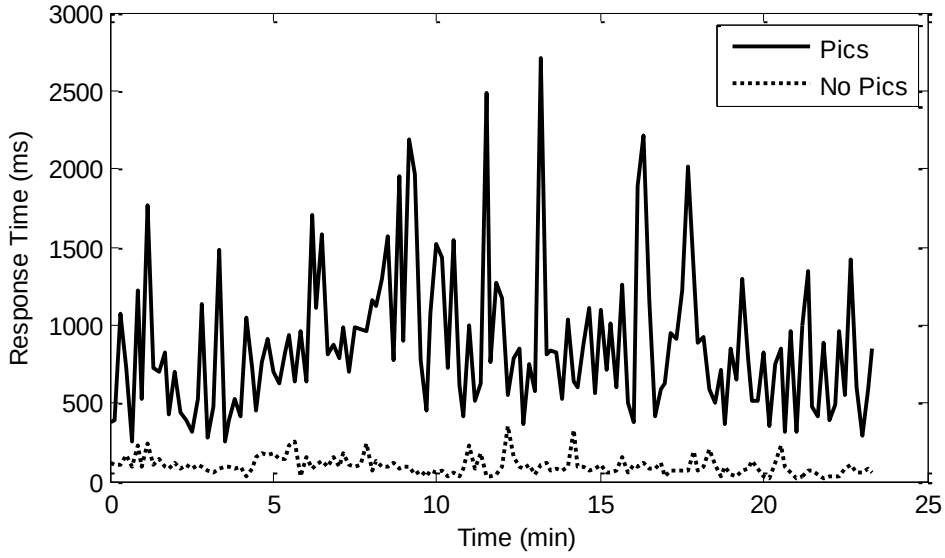


Figure 3-10: C-MART Response Times when pictures, CSS, JavaScript are and are not downloaded

3.7.4. Performance Prediction

Performance prediction schemes typically rely on applications' historic and current workload levels to estimate the resources required to achieve a given QoS level and mitigate violations. For example, [77] estimates resource utilization caused by each incoming request using regression analysis, then determines current resource needs based on current request levels. Such schemes are effective only if the ratio of each type of incoming request can be accurately predicted, as they can be in current benchmarks that use Markov chains to transition between pages. However, as noted in [76], this behavior is not observed in production applications.

Figure 3-11 shows the regression analysis for incoming request rates and database CPU utilization. The request rates were normalized for comparison purposes. The graphs show the regression line bounded by fit lines of one standard deviation. When using RUBiS the database's CPU utilization is highly correlated to the number of incoming requests. However, due to greater variation in page behaviors, the correlation between CPU utilization and the number of requests in C-MART is significantly reduced. The equations in terms of the workload, λ, are:

$$CPU_{RUBiS} = (20.4 \mp 0.9)\lambda + (0.1 \mp 0.6)$$

$$CPU_{C-MART} = (18 \mp 9)\lambda + (26 \mp 8)$$

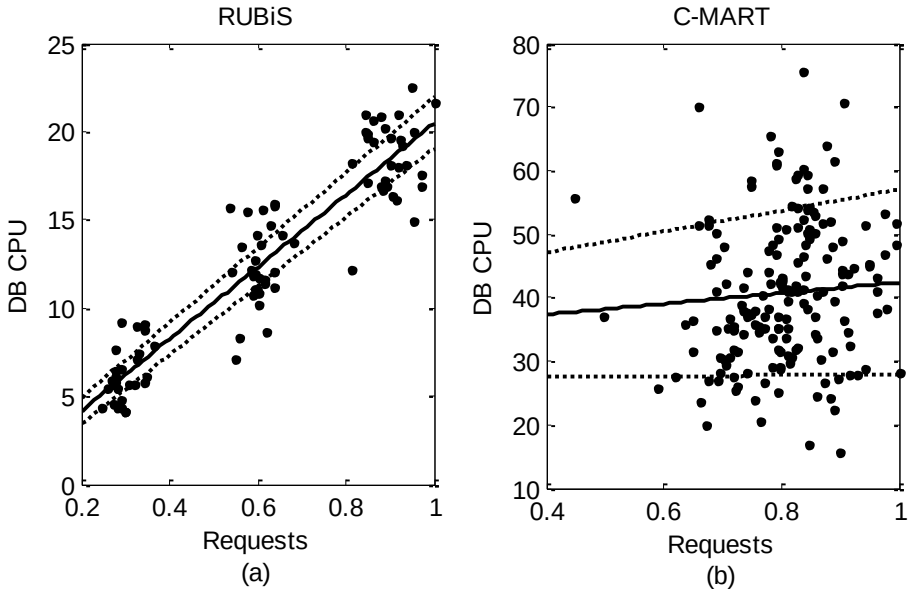


Figure 3-11: CPU Prediction based on workload for (a) RUBiS and (b) C-MART
Regression line shown with one standard deviation

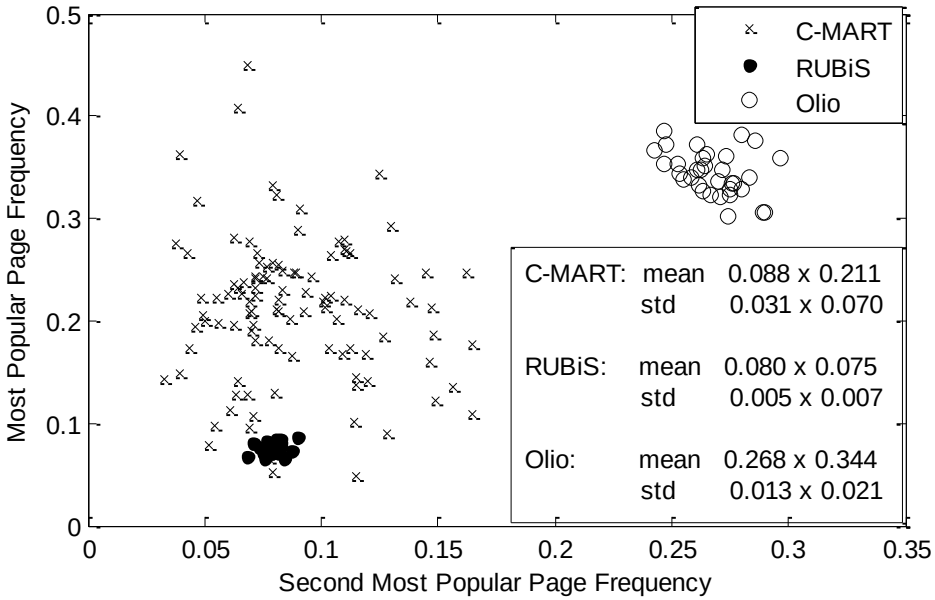


Figure 3-12: Frequency of popular page accesses in different time intervals

The RUBiS equation has little error on its slope and passes through the origin within error. The C-MART slope has a standard deviation half of its actual value, a 1040% increase in error, and the equation greatly misses the origin. This is not nearly well enough defined to be used for analysis. Applying the estimation scheme used successfully in RUBiS to C-MART would result in significant resource under-provisioning and QoS violations.

To further illustrate the static nature of request patterns created by Markov chains, Figure 3-12 shows the ratio of requests for the most popular and second most popular pages for C-MART, RUBiS, and Olio during ten-second periods. The ratio of page requests in RUBiS and Olio is clustered around a single point with a low variance. However, the ratio of page requests in C-MART varies from roughly 10:1 to 2:3 of most popular to second most popular. This closely follows the observed behavior of a production website in [76].

3.7.5. Caching and SQLite

Caching allows for files such as images and CSS to be accessed locally after the first download. Similarly, SQLite can store dynamic data used to populate the webpages. Only small updates in the form of JSON or XML objects need to be sent from the application server to the client on subsequent requests as the client already has the previously accessed data and performs the page rendering itself. Database access and response time become more bursty and varied as the amount of data required to be read is not constant for a given page.

To demonstrate this we examine the download sequence of the Browse page for RUBiS and C-MART, with and without SQLite, when loading the pages in Google Chrome. When first loading the C-MART Browse page with nothing cached the response time is 3.76 s, as JavaScript, CSS, images, and a prepopulation file for the SQLite database are downloaded. The second load of the Browse page for the same client takes only 309 ms as images can be obtained from the cache and only data which has been modified needs to be sent from the application.

For example, a JSON response containing the items listed on the page reduces from 34 kB to 598 B, and a 1.69 MB response that is used to populate the SQLite database on the first access is not required on the second. The first page load contains fifty-one different requests while the second contains only forty-seven. By comparison, the RUBiS Browse page contains the same three requests for every access of the same page. Olio contains more requests than RUBiS (thirty-two on the Home Page) including images and scripts; however the request sequence is identical for each load of the page.

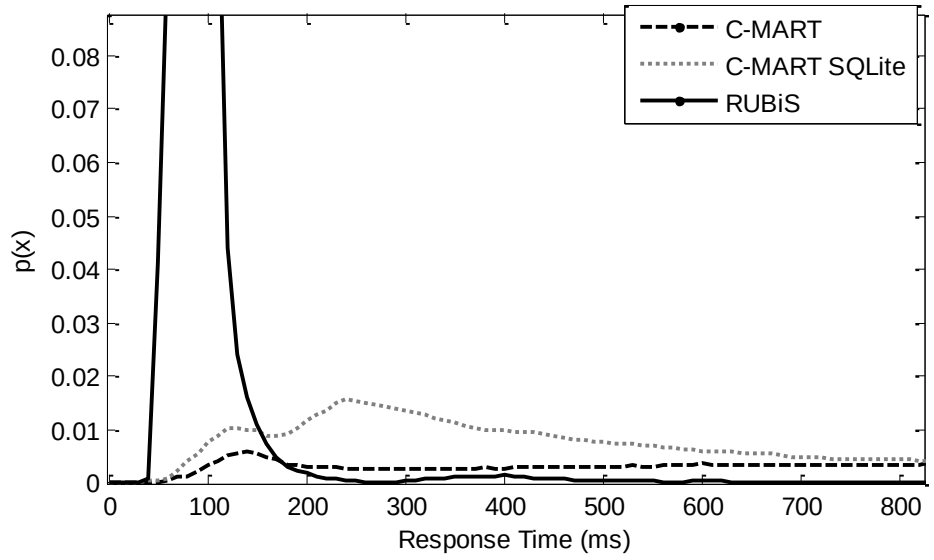


Figure 3-13: Item page response time distributions for C-MART, with and without SQLite, and RUBiS

The use of caches makes profiling based on request type difficult. The increased variability and complexity of page response time distributions results in too much error when profiling using only the distributions' average or median. The distributions of response times for the Item Pages are shown in Figure 3-13. RUBiS has an extremely large peak (that goes off the scale but is not shown for readability purposes) as all Item pages are essentially identical. The non-SQLite version of C-MART has one peak but is more spread out due to the larger differences between pages. The C-MART SQLite distribution is interesting as its peak at 130 ms is larger than the non-SQLite distribution. This is a result of the item data already being stored in the local client database. If there are no updates all that needs to be obtained from the server are the new images. The second peak in the C-MART SQLite version is for when the item is not already in the client database and extra processing is required.

These wider and more complex distributions make it more difficult to determine response time as a function of request type. For the given RUBiS distribution, the response time

is 109 ± 138 ms. The average and standard deviation of the response times for the C-MART non-SQLite and SQLite versions are 4000 ± 7800 ms and 1720 ± 6100 ms respectively. These variations are far too great to be used for a linear regression analysis to relate the response time to the page type.

3.7.6. QoS Measurement

It is common for transaction based applications to measure QoS on an aggregate level over all clients using the average or percentiles of response times. However, in C-MART, clients make decisions based only on their individual QoS, not the QoS received by other clients.

When determining how QoS affects clients it is important to run the workload generator in an open loop mode. A closed loop generator automatically replaces an angry client leaving with a new client, holding the total user level static. This does not accurately represent the loss in profit that would occur from clients leaving the site prematurely. In real applications, the client arrival and departure rates are not directly dependent on one another. Clients leaving due to poor QoS would be reflected in a decline in workload. Using a closed loop generator may give the impression that a resource provisioning scheme is performing well, when it would perform poorly in a production environment by causing many clients to leave the site prematurely.

We run C-MART twice in open-loop mode with the same arrival rate for both experiments. In C-MART the probability that a client leaves the website increases as the response time increases above a threshold value. The thresholds are set to 400 ms and 40 ms in the two respective experiments. The resulting client levels over the duration of the experiments

are shown in Figure 3-14. The clients in the 40 ms experiment have much higher QoS expectations and are therefore more likely to leave when service is slow. This is reflected in the lower client level.

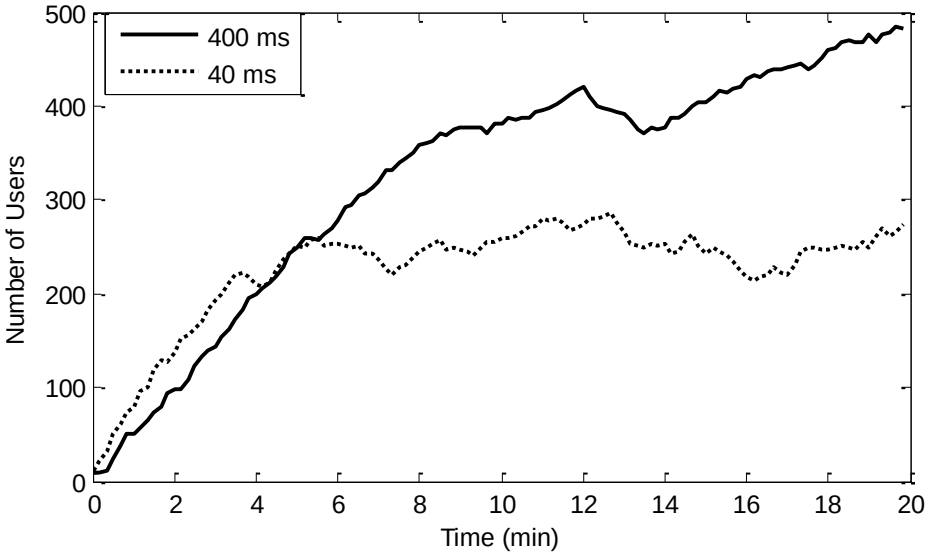


Figure 3-14: User load for different response time expectations with an open-loop client

In Table 3-4 we show clients' average session lengths and the percent of clients that leave unsatisfied due to poor QoS. It also shows how measuring QoS with aggregate response time can suggest much better results than when considering each client individually. When using aggregate response time, clients do not leave as often because the clients receiving bad service are balanced out by those receiving good service. Using this metric would overestimate the effectiveness of a SUT as real world clients are not satisfied by other clients receiving good service; they want the good service themselves. Results for closed-loop clients are also included. Since the unsatisfied clients leaving are automatically replaced by new clients, the client load is

not reduced which keeps the response time high. The closed-loop mode is not an accurate representation of the natural feedback that would occur in such a system.

Situation	Percent of Clients that leave due to poor QoS	Average Client Session Length (s)
400 ms Response Time Threshold, Per-client QoS, Open Loop	19.5%	333
40 ms Response Time Threshold, Per-client QoS, Open Loop	48.4%	201
40 ms Response Time Threshold, Aggregate QoS, Open Loop	19.8%	239
40 ms Response Time Threshold, Per-client QoS, Closed Loop	57.7%	171

Table 3-4: Results of User satisfaction with C-MART QoS

3.8. Conclusion

Existing benchmark applications do not represent modern websites and are inappropriate for benchmarking cloud systems. We present C-MART, a new benchmark application designed to emulate the behavior of modern cloud computing applications. C-MART can dynamically scale to support a large number of clients and has a flexible application design, allowing it to emulate multiple different application architectures. It uses modern web technologies such as HTML5, CSS, AJAX, and SQLite and includes a workload generator that emulates clients accessing the website. It creates unique clients that change their behavior according to page content, history, and QoS. These factors make the clients' behaviors more realistic and increase the variability of servers' utilizations and response times. C-MART's

deployment server allows automatic, simple, and fast datacenter configuration and resource provisioning.

Our results show that existing benchmarks are overly optimistic their evaluation of management systems for cloud environments as they are unable to identify many deficiencies in the SUT. We show that existing workload prediction models could have 1040% greater error than what was previously validated. We also demonstrate how existing management schemes could under-provision applications' resources in 22% of time intervals.

4. Dynamic Resource Controller

In this chapter we present our dynamic resource controller. Our controller aims to achieve SLOs by automatically allocating VMs' resources when they are required. Resources are then taken away and reallocated to other VMs as resource requirements change due to variations in workload. In this chapter we are working as the cloud administrator and controlling the underlying VM architecture of the cloud computing environment. This could be either a public or private cloud. We want to allow our customers to specify the response time that their application should achieve as a service level objective (SLO); for example, to achieve 100ms 90% of the time. This is not a feature that current cloud providers give their customers.

In order to achieve applications' SLOs, each application's resource allocations must be sufficient. In addition, the contention for resources experienced by applications must be limited. In a non-virtualized datacenter, applications avoid performance degradations by being isolated and running on dedicated hardware. However, this typically means low resource utilization levels, resulting in high hardware and energy costs. It is therefore attractive to place applications within VMs to reduce these costs, such as in a cloud computing environment. However, once applications are placed in a cloud environment, they must contest for resources as they are no longer entirely isolated. This can cause applications to suffer from performance degradations.

To ensure applications perform satisfactorily, Virtual Machine Monitors can be set to allocate a certain amount of hardware resources to each VM. However, there are a number of problems with current VM management systems. Firstly, the VM's administrator typically needs

to set the resource allocation levels manually. This requires administrators to monitor the applications' performance, and set each VM's resource allocation in the VM management system. This task is made more difficult if the VMs' resource requirements frequently change. Secondly, VM resource allocation levels only guarantee that a VM will receive a certain share of a resource. They do not provide any application-level performance guarantees. This can lead to lower hardware utilization levels, as administrators will typically over-provision resource allocations to ensure satisfactory performance. Lastly, administrators must manually set the utilization levels at which VMs will be migrated to and from hosts. This can again lead to lower hardware utilization as migration thresholds must be set low enough to ensure application-level performance does not suffer due to high resource contention.

To address these problems, our controller system monitors application-level performance and automatically allocates VMs the minimum level of resources they need to meet an application-level SLO guarantee. Our system works by monitoring the applications' performances at various user, resource allocation, and resource contention levels. Resource contention occurs on a host when multiple VMs require the use of the same resource. Once our system has multiple readings at different values, it can interpolate the minimum resource allocations needed for the application to achieve a certain response time.

Figure 4-1 shows the basic flow of information in the management system. The process starts by monitoring an application's response time and the level of resource contention on each host where one of its VMs resides. The management system then chooses a predetermined model that it thinks best describes the performance characteristics for the application. The

models are a set of precalculated response time surfaces with various shapes. Initially, the model chosen will be extremely inaccurate due to lack of data. However, as further data points are collected the model describes the application's performance more accurately. The shape of the model is stretched based on differences between the readings in the model and the currently observed resource contention levels. Missing data points in the model are interpolated from the data that is available. The minimum resource allocation levels that allow the application to meet its response time target are then found in the interpolated model. Lastly, the resource allocations are set on the hosts, and the hosts wait to take a new reading to report to the management system.

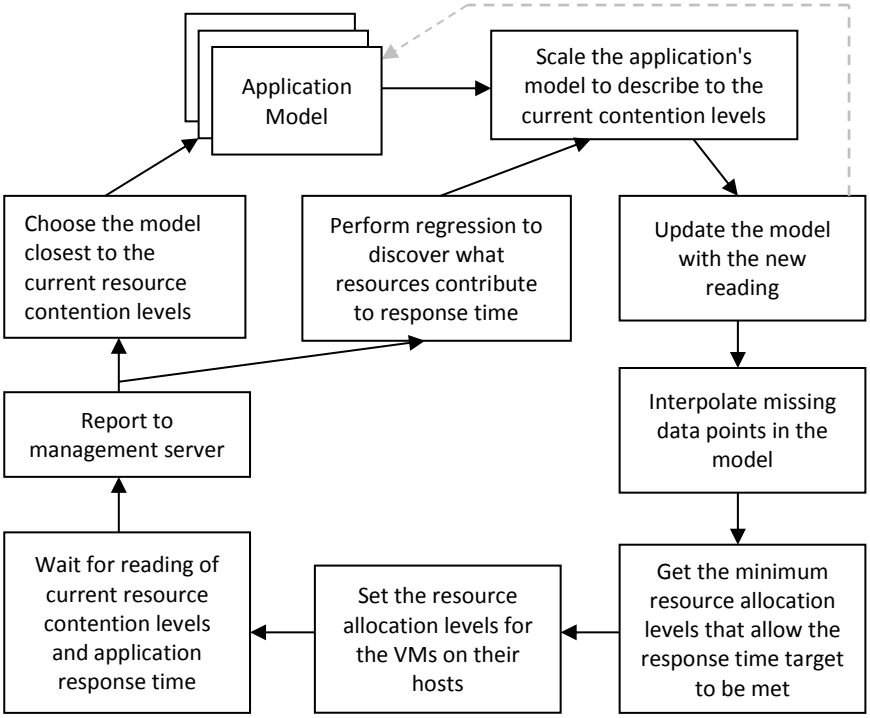


Figure 4-1: Management system flow

Applications' performance models are created automatically by analyzing the performance observed at the various resource allocation levels. Although such models could contain millions of potential data points, we have found that a model with only 10's of data points allows accurate performance predictions.

As cloud based applications are typically multi-tiered, our system allows for this. It sets the resource allocations at each tier, such that the total end-to-end response time experienced by the user is below the SLO target. This allows a cloud customer to configure a single SLO value for an entire application stack. This is in contrast to current management systems, where the resource allocation must be configured manually by an administrator at each tier without knowing the effect on end-to-end response time.

4.1.1. Monitoring

To collect the data we need for our management system we record the application's response time at its first tier, as shown in Figure 4-2. We monitor the response time using a custom written Apache HTTP server module that monitors response time per page. Throughput based applications can be monitored in a similar fashion, with throughput per time period recorded rather than response time. While monitoring response times at each individual tier could provide a more accurate model, such monitoring would incur a significant overhead. Additionally, monitoring at intermediate tiers does not always reflect the overall performance characteristics experienced by the end-users.

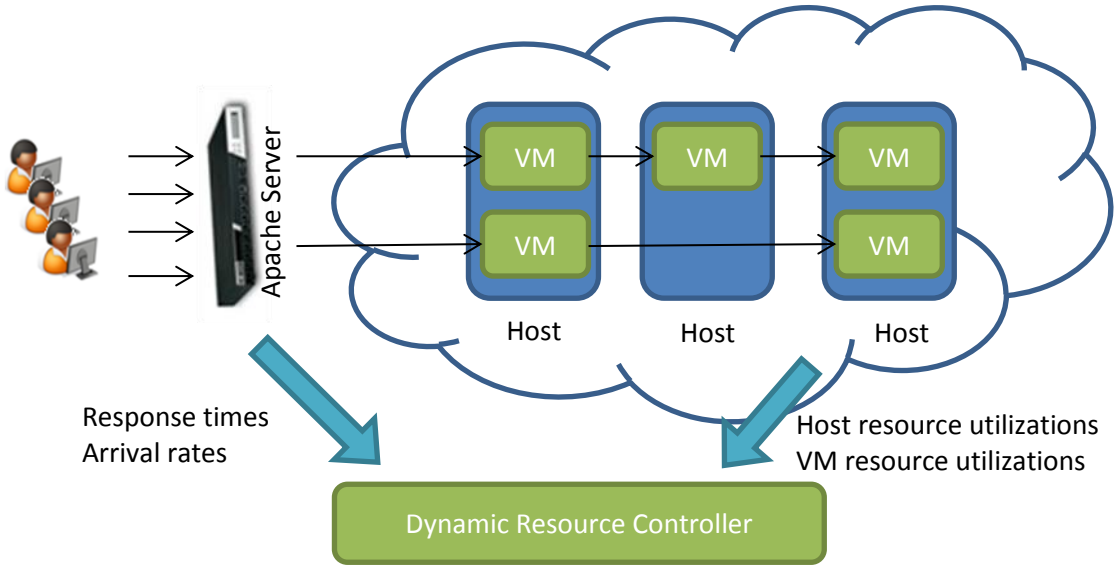


Figure 4-2: Response time monitoring

The data we capture are the applications' per page total response time distributions, CPU utilization, and storage and network throughput. All of the data are captured outside of the VMs, thereby not requiring a client to be inside the VMs. To allow our system to react quickly to changes in user workloads, we take a reading every 10 seconds. This period could be increased or decreased as needed, depending on the system being controlled. However, we have found that 10 seconds is a good balance between a response system and a sufficiently low monitoring overhead.

After the data is captured, it is passed to our server and added to the application's model. The model then interpolates the resource allocations that each VM should receive to meet a specified response time and chooses the minimum value. These resource allocations are then set on each host so that each VM receives the amount of resources calculated by the model, as shown in Figure 4-3.

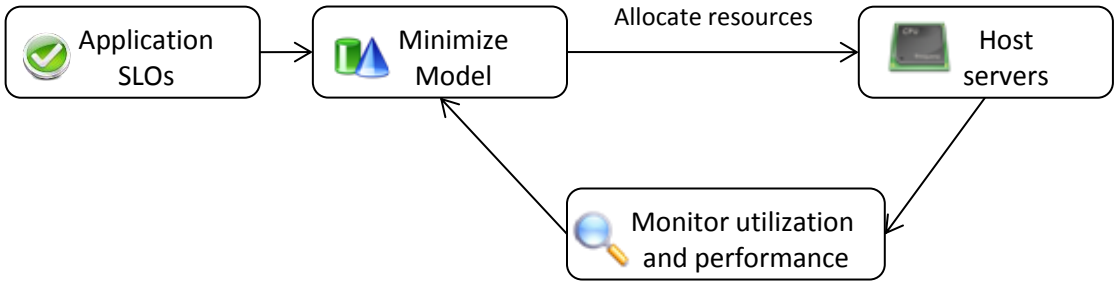


Figure 4-3: Control loop

4.1.2. Model Interpolation

As there are potentially millions or billions of possible data points to describe the performance characteristics of each application we cannot capture the state space of an application before making resource allocation decisions. Instead, to predict the required resource allocation levels we must identify trends in the data. To demonstrate this, Figure 4-4 shows the effect of CPU contention on the web tier of TPC-W [11]. The CPU contention is the total CPU utilization minus the amount used by the TPC-W VM itself; i.e. the utilization not related to TPC-W processing. As shown, the response time curve follows an exponential distribution accurately. As this data closely fits an exponential distribution, few iterations of the controller and data points would be needed during run time to interpolate and estimate resource allocation values.

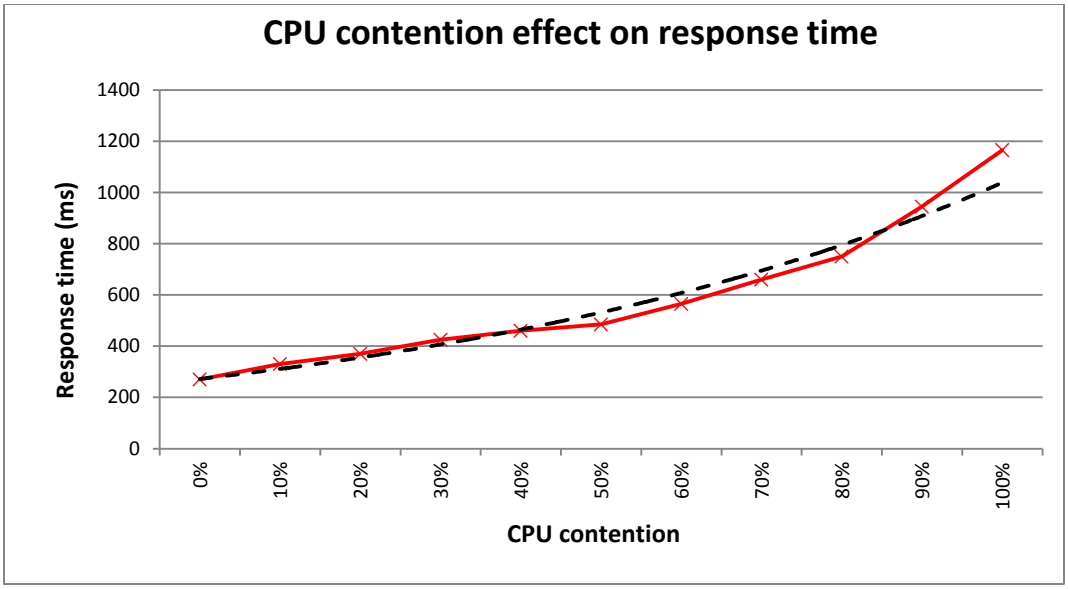


Figure 4-4: CPU contention and response time degradation

To further demonstrate this, Figure 4-5 shows the response time of TPC-W as the web tier has its CPU allocation changed from 10% to 100% in 1% increments. We cap resource allocation levels at minimum of 10% in our system as we have found that response times quickly approach infinity (the website crashes) for extremely low resource allocation values. This is due to OS and application housekeeping requiring a minimum amount of resources. As shown, for 45%-100% CPU allocation the response time for all four contention levels can be roughly predicted by the same linear function. For allocation values less than 45%, each contention level follows its own steeper linear function. This occurs as we are only displaying the resource contention and resource allocation for the web tier in the figure. When allocated over 45% of the hosts' CPU the web tier is no longer the application's performance bottleneck. Allocating additional resources over 45% to the web tier at this point provides no improvement in the end-to-end performance of the application.

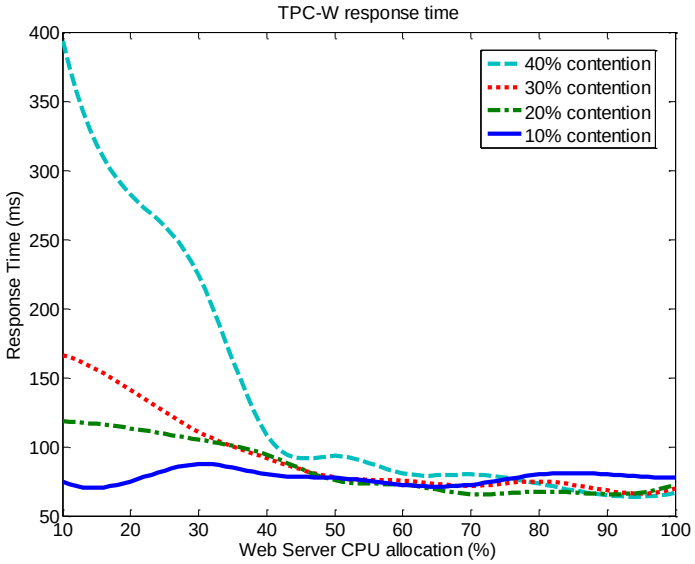


Figure 4-5: TPC-W response time with proxy and web server set at 80% CPU allocation

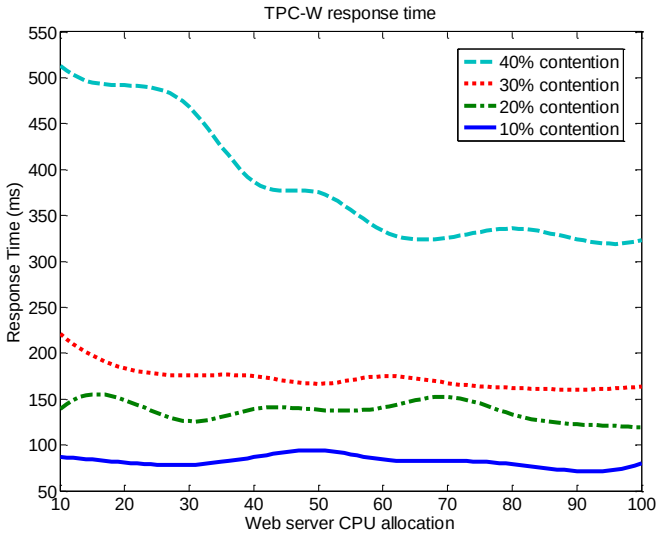


Figure 4-6: TPC-W response time with proxy set at 80% CPU allocation

Let us assume that the customer that is hosting TPC-W in our cloud environment has specified a 100ms SLO. Figure 4-5 suggests that we allocate the web tier 45% of the CPU share if the CPU contention on the host is 20% or above. However, this only considers a single tier of the application. Figure 4-6 shows the experiment – response time curves when the web tier's CPU allocation is changed from 10% to 100% – except we have reduced TPC-W's proxy VM to 30% CPU allocation on its host. In this situation, there is no way to achieve the 100ms response time target if the contention for TPC-W web tier is more than 10%. This is because the proxy tier has become the application's bottleneck, so allocating more resources to the web tier will not significantly improve the response time. This clearly demonstrates that to minimize the VMs' resource allocation levels the model must include every tier of the application as a dimension.

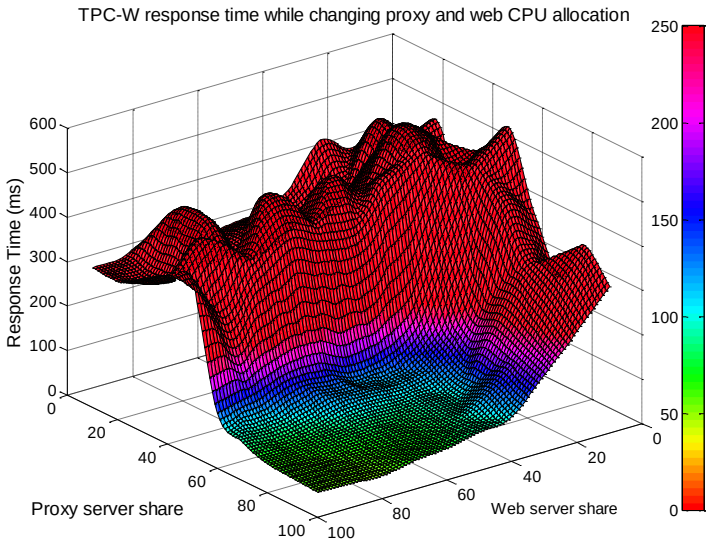


Figure 4-7: Proxy and Web tier CPU allocation response times for 40% CPU contention

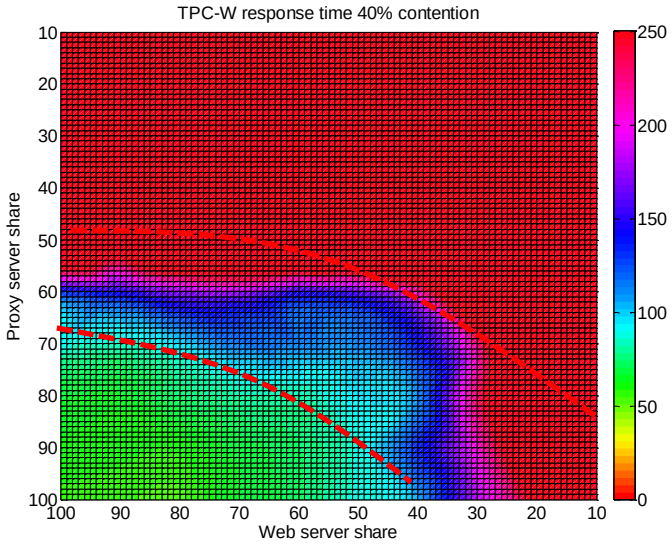


Figure 4-8: Proxy and Web tier CPU allocation response times for 40%CPU contention

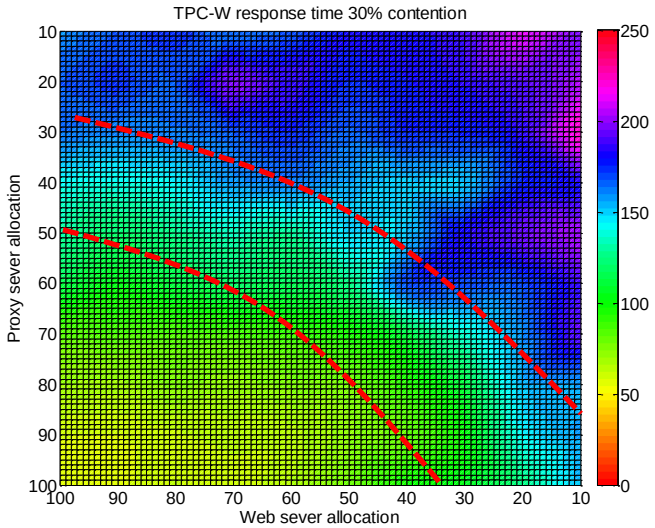


Figure 4-9: Proxy and Web tier CPU allocation response times for 30% CPU contention

Figure 4-7 shows the surface plot for the TPC-W proxy and web tiers with 300 active users and 40% CPU contention on each host. It should be noted that our management system uses data from every application tier and from multiple hardware components. We limit to displaying CPU allocation as it is the primary factory effect response time, and displaying graphs with more than three dimensions is difficult.

Although Figure 4-7, Figure 4-8 and Figure 4-9 contain hundreds of data points to show the complete resource allocation to response time model, the runtime model does not require this much data. If, for example, the user has set 150ms as the SLO target, the resulting model will collect data points around that response time, but very few data points for the rest of the model space. For example, in Figure 4-8 and Figure 4-9 the model will mostly need to record data points between the dotted lines. In addition to having to store less data points, being able to characterize the application with fewer data points helps the model converge and adapt to changes more quickly.

4.1.3. Dimensional Reduction

As there are potentially millions, or even billions, of resource contention combinations, it is infeasible to keep a model for every combination we encounter. Instead, we keep a subset of models, and scale the response time values to fit the current contention levels. To achieve this scaling we use the same data used in the resource allocation to response time models (Figure 4-7, Figure 4-8 and Figure 4-9), but instead interpolate contention to response time for a given resource allocation level. We use piecewise multiple linear regression to estimate the value that each point in the model should be scaled by. When we are estimating resource

allocation values for a resource contention level that we do not have a model for, the management system needs to choose the model that most accurately represents the current resource contention levels. The model we choose to scale is the one with the smallest Euclidean distance from the current resource contention levels.

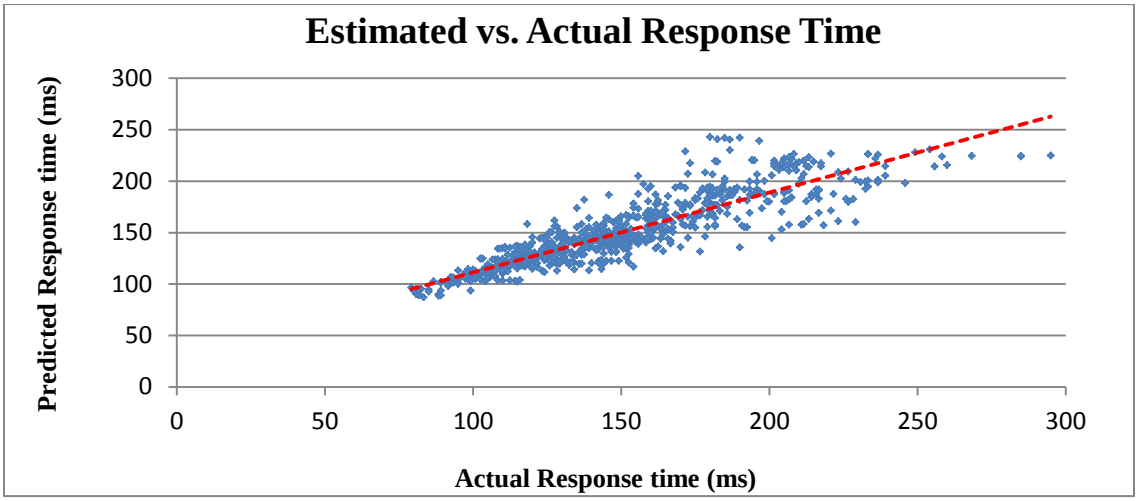


Figure 4-10: Estimated and Actual TPC-W response time

Figure 4-10 shows the actual response time for TPC-W and the estimated response time calculated using the regression coefficients. The data shown is a subset of data points where the CPU contention is between 10% and 40% for each tier. As can be seen in Figure 4-10, the estimated and actual response times are highly correlated, as would be expected given the fit of the data shown in Figure 4-4.

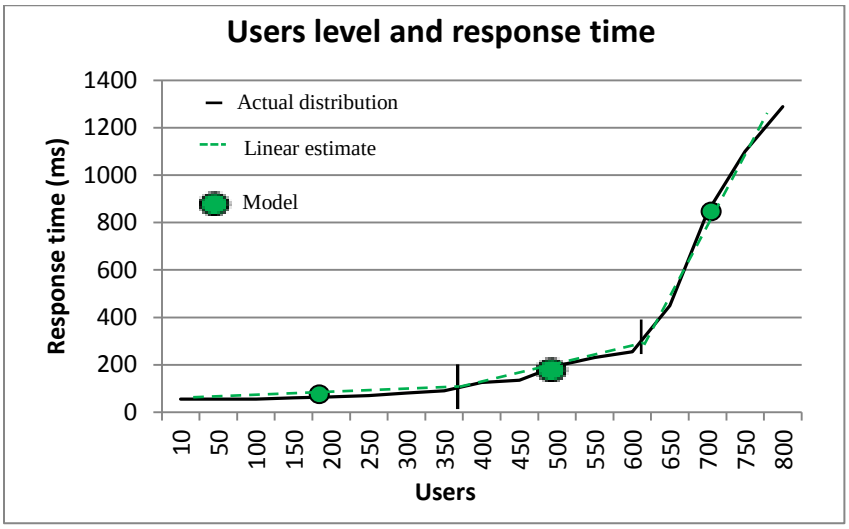


Figure 4-11: Response time increase vs. user level

Figure 4-11 shows how TPC-W's response time increases as the number of users increase; in this experiment there is 10% CPU contention on each of the hosts where the TPC-W VMs are placed. As can be seen, the response time increases exponentially with the number of users. This can be accurately represented by linearly scaling three copies of an application's resource allocation model.

Figure 4-12 shows the degradation in TPC-W's response time at various CPU contention levels. The response times shown are when TPC-W's web tier is allocated either 50% or 10% CPU. At 50% CPU share allocation the CPU contention has little effect on the response time. This is because the web tier receives CPU cycles very frequently, and is not the application's bottleneck. At a 10% CPU share allocation the response time quickly degrades with almost a 50% increase in response time with a 10% increase in CPU contention. Even though the CPU had over 40% free cycles, the web tier does not receive its cycles promptly enough. This causes it to become the bottleneck tier and results in degraded response time.

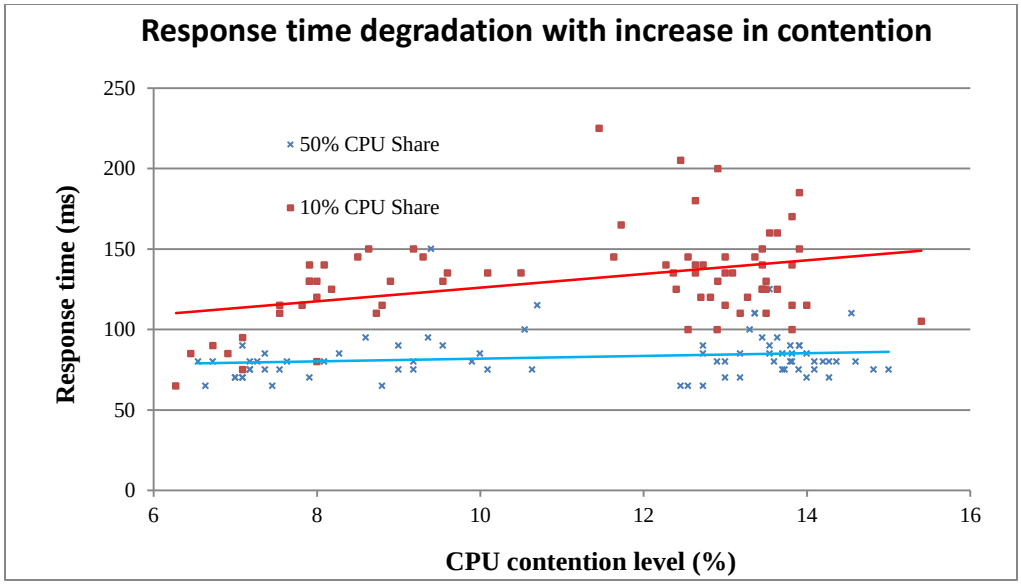


Figure 4-12: Regression values used to stretch a model

To estimate an application's performance at a previously unseen resource contention level we use the fit for previously observed data. For example, from Figure 4-12, if we want to estimate response time 20% CPU contention and 50% CPU share, it would be 73 + 1.1 * 20 = 95ms. If our SLO target is 100ms, we would know that we could place the web tier on a host with 20% CPU contention if it could receive 50% of the CPU share allocation. However, if the host only had 40% CPU share allocation remaining, the estimated response time would be 80 + 1.15 * 20 = 103ms. Therefore, we would not expect that we could place the web tier on that host.

4.2. Implementation

Our Dynamic Resource Controller is implemented using various technologies. We must utilize multiple technologies as our system touches many different areas of the datacenter. Figure 4-13 provides an overview of the technologies we use and where in the system they are utilized.

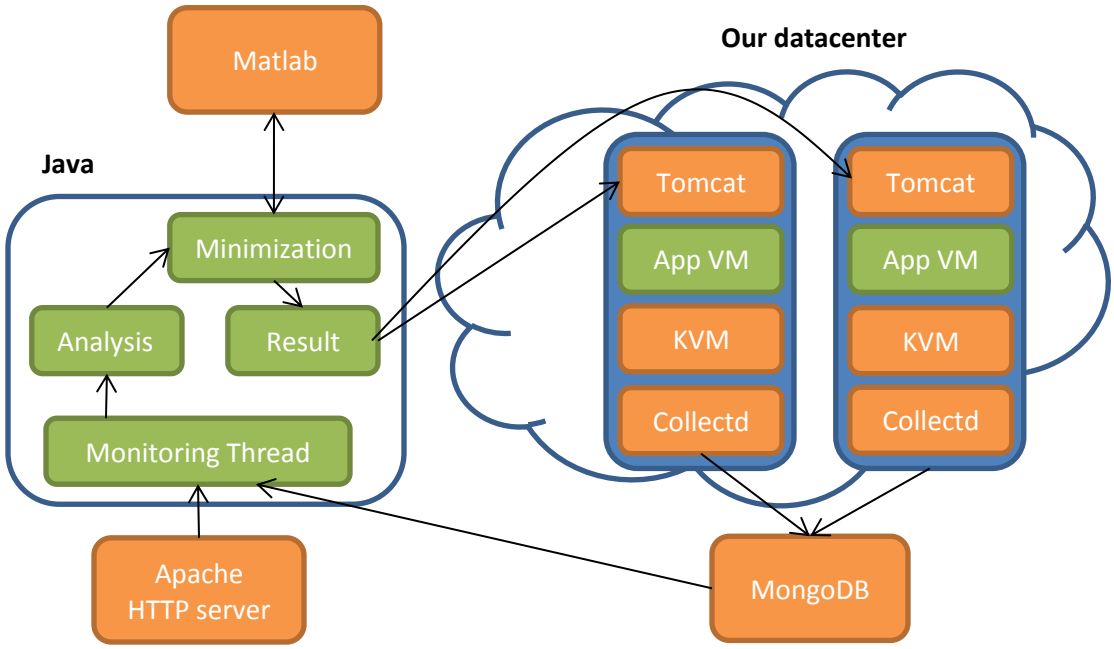


Figure 4-13: Implementation of Dynamic Resource Controller

The majority of the logic for our Dynamic Resource Controller is implemented in Java, as shown in Figure 4-14. A brief overview of the function of each external component is as follows:

Apache HTTP server: Monitor incoming requests' response times and load balancing

KVM: The Hypervisor that allows us to host VMs

Tomcat Server: To remotely control KVM's resource allocations

Collectd: Monitor the resource utilizations of hosts and VMs

MongoDB: Distribute the monitoring's reporting overhead, and provide redundancy

Matlab: Perform convex optimization of VMs' performance models

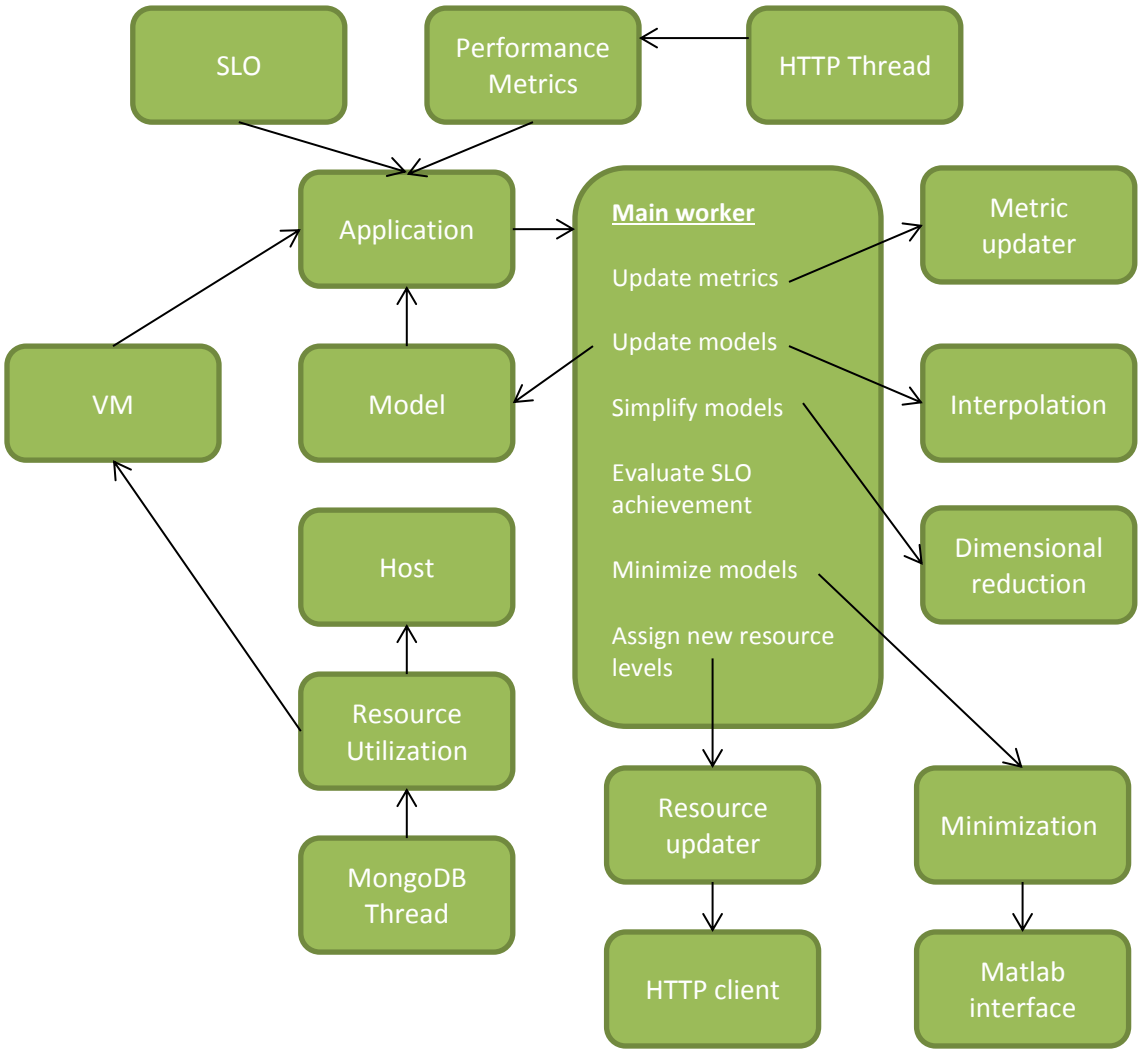


Figure 4-14: Java controller implementation

When the controller first starts it reads a configuration file that provides it with enough information to begin monitoring applications. The file contains information about the applications' VMs, the URL via which the application is accessed, and the SLOs. It also includes datacenter information, such as hosts' addresses, MongoDB's address, and where to obtain monitoring data. As the controller initially has no information about applications' performances, it first attempts to learn about the applications' behaviors. Figure 4-15 shows the timeline view of the controllers' behavior.

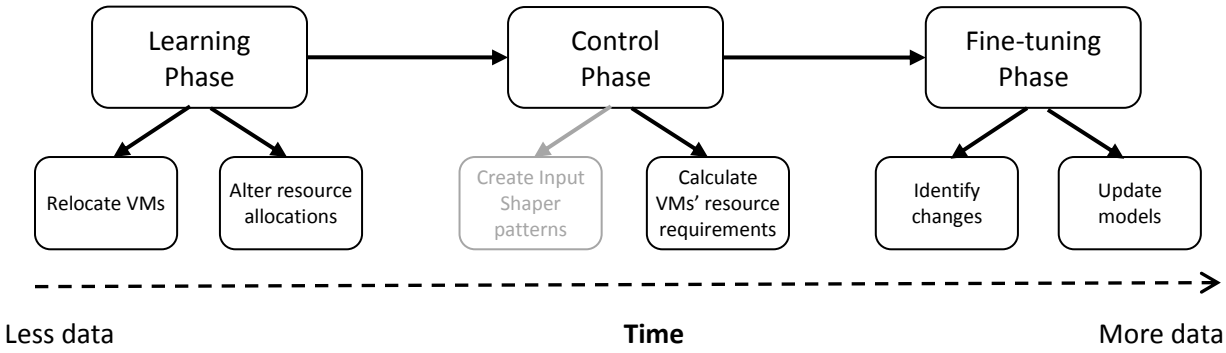


Figure 4-15: Timeline view of application control

4.2.1. Learning Phase

Initially our controller may not be able to achieve applications' SLO. Applications may be placed such that they cannot achieve their SLOs due to lack of available resources. Even if there are sufficient resources on a host to allow an application to achieve its SLO our controller does not yet know that value. To gain data on an application's performance characteristics the controller alters the resources allocated to the applications at each tier in turn to observe what happens. This allows the controller to begin constructing a model of the applications' performances. Table 4-1 shows the data that is collected about each application.

I^k	A vector of the CPU, Disk and Network utilization of a physical host *k*
U_i^j	A vector of the CPU, Disk and Network utilization of a VM *j* that belongs to application *i*
$C_i^{j,k}$	The resource contention experienced by VM *j* that is located on host *k*, $I^k - U_i^j$
W_i^j	The number of requests received by VM *j*
A_i^j	The resource allocation level of VM *j*
R_i	The observed response time of application *i*

Table 4-1: Data collected relevant to applications' performances

Initially, an application's performance model will be sparse with data; we assume that its performance follows a truncated Pareto distribution. We choose this distribution as previous works have shown that heavy-tailed distributions commonly describe the tail of applications' performances [78] [79]. To fit the data to a truncated Pareto distribution we must estimate its three parameters: shape, scale, and maximum. The maximum response time that a request can receive is the timeout value of the server processing the request. After this time the client will receive an error message from the server. We use this as the maximum value. To estimate the scale of the distribution we use the distribution's mode value. We choose this value as applications' performances typically begin to tail off after the most frequent response time. It is possible that some applications may exhibit uniform response time, or some other distribution vastly different from a truncated Pareto. However, during the leaning phase we must make some assumptions and consider the most common case. Also, as the truncated Pareto distribution is heavy-tailed we are likely to overestimate applications resource requirements,

rather than underestimate estimate. To estimate the shape parameter of the distribution we use the truncated Pareto's maximum likelihood estimator, as shown below.

$$\gamma = timeout\ time$$

$$\beta = mode(X)$$

$$\frac{n}{\tilde{a}} + \frac{n\frac{\beta}{\gamma}^{\tilde{a}} \ln\frac{\beta}{\gamma}}{1 - \frac{\beta}{\gamma}^{\tilde{a}}} - \sum_{i=1}^{n}[lnX_i - ln\beta] = 0$$

Figure 4-16 shows the log-log plot of the response time of TPC-W and the response time predicted by fitting a truncated Pareto distribution. As can be seen, the truncated Pareto distribution fits the data well and has a coefficient of determination value of 0.99.

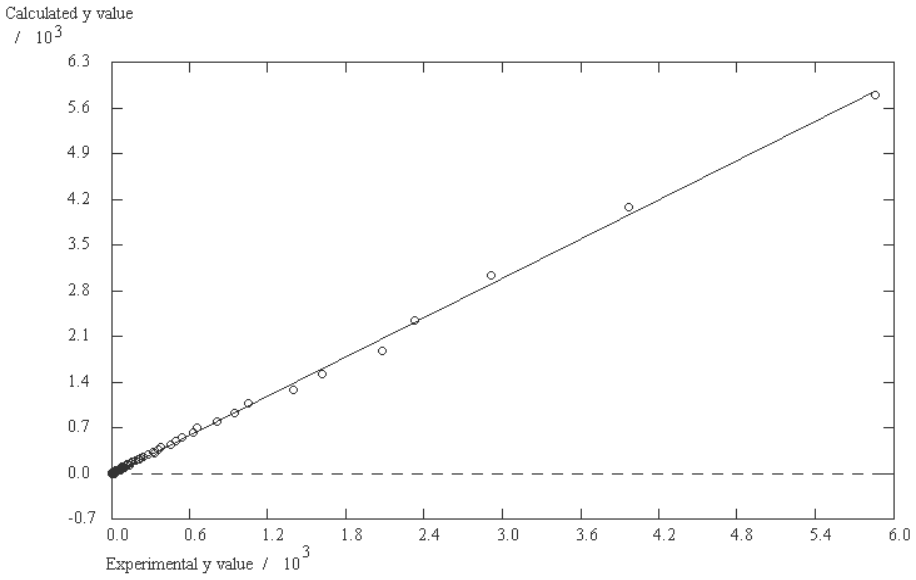


Figure 4-16: Log-log plot of actual and expected application response time

To ensure applications' resource allocation are estimated as accurately as possible, in the control phase each application must have been observed under a number of different environmental conditions. For example, we would like to know how the application performs when there is 50% CPU contention and when there is 10Mb/s disk Input/Output. To aid with this we created a simple load generating VM. The Dynamic Resource Controller can launch the load generating VM on a host and then control its resource consumption via HTTP requests. The controller initially sets the resource consumption of the load generating VM to low. This allows the application to achieve its SLO during most of the profiling period. The controller drives up the resource utilization of the load generating VM until it finds the application's maximum resource contention limit; i.e. at the point where the application begins to fail its SLO.

To monitor each VM's resource utilizations we use the Linux collectd tool connected to MongoDB. Collectd connects to the libvirtd process to collect data about VMs' resource utilizations such as CPU utilization, Disk I/O, etc. We use MongoDB as its scalability, performance, and redundancy suit our needs well. Using collectd we can collect hundreds of metrics for each VM and host; this results in thousands of collection reports per second. Distributing the load over multiple MongoDB hosts prevents overload at our monitoring tier. Also, as MongoDB is configured with multiple shards it provides additional redundancy.

4.2.2. Control Phase

Once the applications' models have data for numerous resource allocation, resource contention, and incoming request rates, we stop exploring their state space and allocate them only the resources required for them to achieve their SLO. To calculate VMs' resource

requirements we perform a convex optimization on their performance models. The performance models are convex as applications' performances do not decrease if they are allocated additional resources. Similarly, applications' performances do not increase if they experience greater contention for access to shared resources. The optimization calculates the VMs' minimum resource allocations that allow the applications to achieve their SLOs. The optimization performed is:

Minimize:	$1X^TA$	
Subject to:	$X >= 0$	*no negative resource assignments*
	$X^TA <= 100\%$	*host utilization <=100%*
	$1^TX >= 1$	*must choose a solution*

Where A is the acceptable resource allocation levels as indicated by the applications performance models, and X is the chosen solution.

To perform the optimization we outsource the operation to Matlab. We utilize the CVX library created by Stephen P. Boyd [80]. A Java connector exports the performance models as matrices to Matlab, which returns a matrix of VMs' resource requirements. If there is not a solution where all VMs are satisfied the closest non-solution is returned. We can check for overloaded hosts by calculating each host's total resource allocation, $\sum_{i=0}^{n} VM_{allocation}^{i} \ on \ host \ k$. If the total allocation is greater than 100 then the host is overloaded. In this scenario we have calculated whether all VMs can achieve their SLOs, and if they cannot, we have identified the optimal host to migrate a VM from.

To set each VM's resource allocation we connect to each host over HTTP. Each host runs a simple web application that we created to allow us to alter VMs' resource allocations in the KVM Hypervisor.

4.2.3. Fine-tuning Phase

Once applications are achieving their SLOs the Dynamic Resource Controller needs to tweak applications' resource allocations rather than perform dramatic datacenter rearrangements. For example, if a suitable resource allocation for the VMs (App, DB) is either (10,90) or (90,10) we do not want the chosen solution to flip-flop between the two. If it does there could be a chain reaction of resource allocation changes that results in all VMs' resource allocations changing wildly during each control period. To rate limit the amount of resource allocation change we only pass part of applications' performance models to Matlab, limiting the number of solutions that are available to choose from. For example, if the rate limit is 5% and a VM's current CPU allocation is 40% we only pass the models' values between 35% and 45%. If no valid solution is found within the rate limited region we repeat the process but double the allowable change. If again no solution is found then we consider that we are no longer fine-tuning allocations, but are instead creating an entirely new solution. When large resource allocation changes occur it is more likely that applications will fail their SLOs as there is already transient load within the system which may that have to be processed on now underprovisioned VMs.

In addition to variations caused by workload and VM placement changes, applications' underlying performances themselves can also change; for example, as a result of a code change

or the buildup of data over time. To identify this in the fine-tuning phase we consider that a VM's underlying performance has changed if its observed performance is further than two standard deviations from the expected value for three consecutive control periods. If such a change is identified the application is sent back to the learning phase.

4.3. Experimental Setup

4.3.1. Infrastructure

Our experiments are performed on flat local area network using commodity hardware. This is similar to the hardware found in cloud computing environments. The hosts' operating system is Fedora 12 with Linux kernel 2.6.31. We use KVM as our hypervisor. The VMs' hosts consist of three nodes with a tri-core 2.1 GHz CPU, and 4GB RAM. The test client nodes consist of two hosts with quad-core 2.66 GHz CPU, 4GB RAM. The storage node contains a dual-core 2.8 GHz CPU, 4GB RAM.

The network topology we use is two flat-networks each with one switch: the user data network and the management network. This is a typical setup for cloud computing environments. Each physical host has two network interface cards (NICs). One NIC is connected to a user data network using a 24-port Gigabit switch. The user network carries all of the user workload and benchmark traffic. The other NIC is connected to a management network using a separate Gigabit switch as shown in Figure 4-17. The management network carries management-related commands and network attached storage traffic for the VMs' virtual disk images.

The storage system is hosted on two-spindle RAID-0, 2TB, 7200rpm hard disks. The storage server exports an NFS share. All virtual machine images are served from this location. To ensure network storage was not the bottleneck in our system, we benchmarked the network storage and found it more than capable of handling all of the VMs' disk traffic.

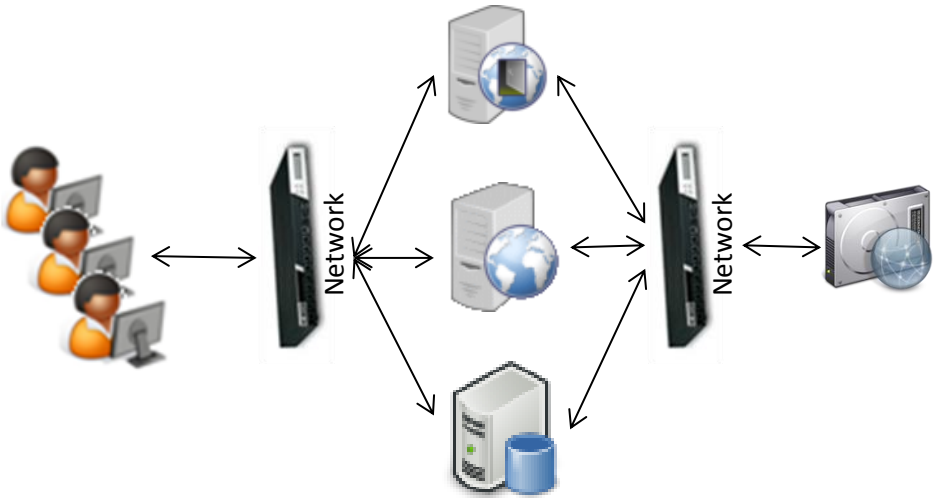


Figure 4-17: Test bed setup

4.3.2. Workloads

To test our system we use the TPC-W benchmark suit [11]. It consists of an Apache web proxy front-end, a Tomcat application server, and a MySQL database back-end. There are 15 types of page requests. The benchmark client is a closed-loop client which simulates multiple users concurrently accessing the server. TPC-W's performance is measured based on response time for each action performed. We choose TPC-W as it is simpler to control than the C-MART benchmark described in Chapter 3, but provides sufficient data to demonstrate our dynamic

resource controller. The Input Shaper is also required to achieve satisfactory results with C-MART, as described in Chapter 5.

4.4. Results

In this section we discuss the results from our dynamic resource controller. We test our system by running the TPC-W benchmark with each of the application's tiers on a separate host. Each host also contains another VM running an Apache web server hosting computationally intensive web pages. The additional VMs are used to create resource contention on the hosts. They represent other applications that would also be running on the host in a cloud computing environment. The number of requests per second to each Apache server was varied throughout the experiments to change the resource contention levels. This mimics the changing workloads of other customers' applications.

4.4.1. Meeting SLO Target

Figure 4-18 shows the resulting response times of TPC-W when the resource allocation levels are set manually and when they are controlled by our controller. Our dynamic resource controller results are labeled "SLO", the manual resources set are labeled with percentage CPU allocation. When the resource allocations are set manually, each tier receives the same resource allocation on each host. For example, in the 50% resource allocation experiment, each tier has a fixed 50% resource allocation throughout the experiment.

As can be seen in Figure 4-18, by using our system TPC-W's response time closely follows the SLO target that is set. It is expected that the response time will oscillate above and

below the SLO target because in these experiments our system attempts to make the median response time equal to the SLO target. It is also evident from Figure 4-18 that the response time when using our system is usually faster, rather than slower, than the SLO target, and therefore averages to faster than the required SLO value. This is due to the resource allocation optimizer being cautious in its estimates. This is a conscious design decision, as a system that constantly over performs is more useful than a system that constantly under performs.

It can also be seen in Figure 4-18 that setting the resource allocation levels manually does not always produce a consistent response time. This is because resource contentions may increase over time, but the resource allocations do not. When the CPU resource allocation is set to 50%, TPC-W's response time is better than our controller's 150ms response time target for a long period. However, at time period 480, a 50% resource allocation is no longer sufficient to continue providing that acceptable response time. By dynamically setting resource allocation our system can keep providing the same response time despite the CPU contention increase.

Test	RT average	Resource allocation average	Apache VM average
SLO = 100ms	89ms	48%	125ms
SLO = 150ms	127ms	35%	107ms
50% resource allocation	150ms	50%	120ms
10% resource allocation	355ms	10%	83ms

Table 4-2: Response time for TPC-W and contention workload

As can be seen in Table 4-2, despite the 50% resource allocation test having a faster response time for a longer period of time than the SLO 150ms test, its final average response time is greater. Additionally, the SLO 150ms test uses on average 15% less resources to achieve this faster average response time. As TPC-W uses less resources in the SLO 150ms test, the

Apache workload on the host receives a greater share of resources, thus reducing its average response time from 120ms to 107ms. This is because the optimizer does not needlessly overprovision TPC-W, allowing the host scheduler to allocate remaining resources as needed. This shows that dynamically setting the resource allocation levels can not only guarantee a specified response time, but is also a more efficient use of resources. In this case, both applications have benefited from faster response times, despite our system only guaranteeing one of them.

Comparing the two tests with the closest resource allocation levels, we find that our dynamic resource allocation helps achieve a faster average response time while using overall fewer hardware resources. Even excluding the final 120 readings, where the 50% allocation test performed poorly, our dynamic resource allocation still performs faster, with an average response time of 89ms vs. the static allocation average of 106ms.

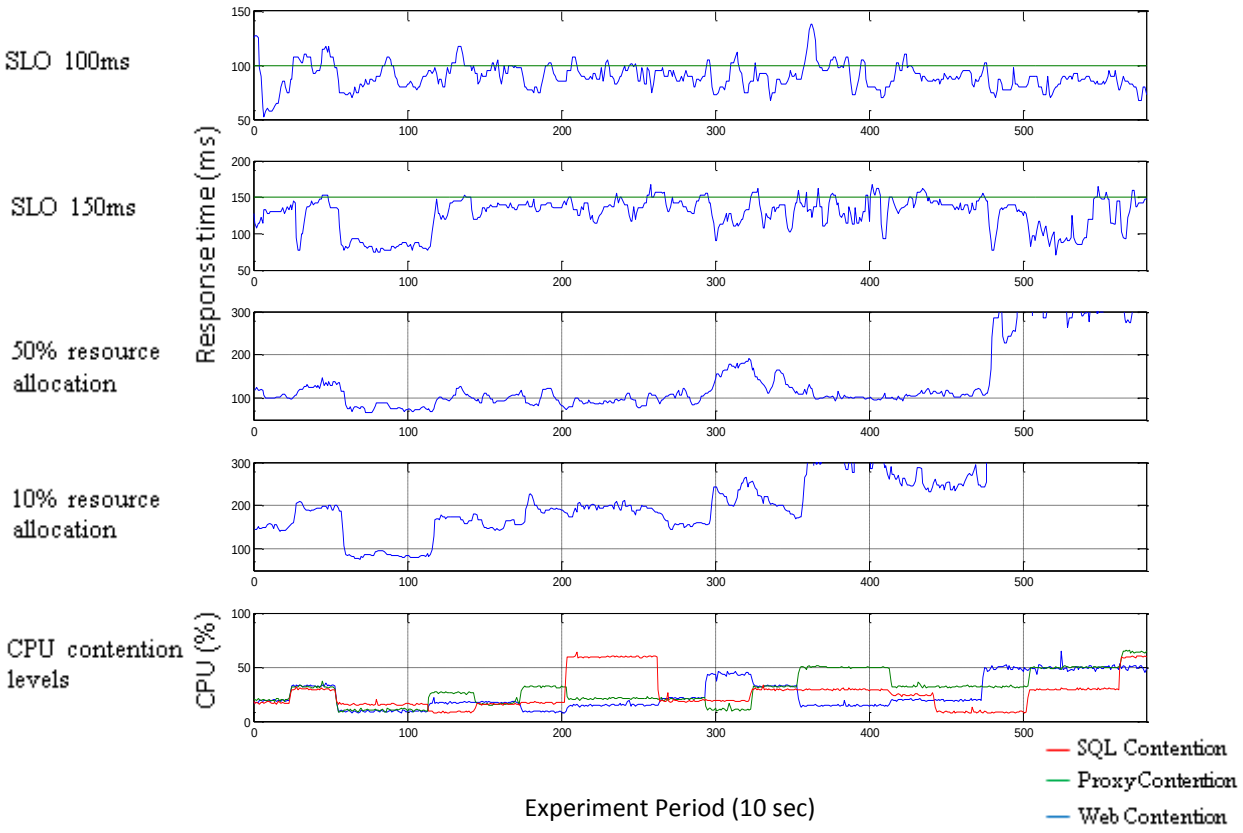


Figure 4-18: Response time results for dynamic and static resource allocations, changing CPU contention

4.4.2. Resource Allocation

Figure 4-19 shows the resource allocation levels that TPC-W was given by our controller for the SLO 100ms and 150ms tests. The other two tests remain at 50% and 10% allocation levels throughout and are not shown.

At time period 200 it can be seen that the CPU contention on TPC-W's SQL VM's host jumps 40%; however, the resource allocation only increases roughly 10%. This shows an advantage of modeling and predicting the application's performance over a more simple

resource control scheme, such as increasing the resource allocation by a fixed factor of CPU contention. Our controller's regression analysis identifies that the CPU contention on the SQL VM's host does not cause large increases in response time. Therefore, when a model is used to predict the resource allocations for the new contention level, the scaling factor is low. This is in contrast to time period 110, when the CPU contention on the web server VM's host increases by 10%. In this case, the resource allocation increases by 20% in the SLO 150ms test and by 30% in the SLO 100ms test. This is because the model has correctly predicted that increased CPU contention on the web server VM's host will cause an increase in response time and has scaled the resource allocation model accordingly. We can see that our control system predicted the correct resource allocation increases in both cases, as the response times for the SLO tests in Figure 4-18 both remain at the configured SLO level at time period 110.

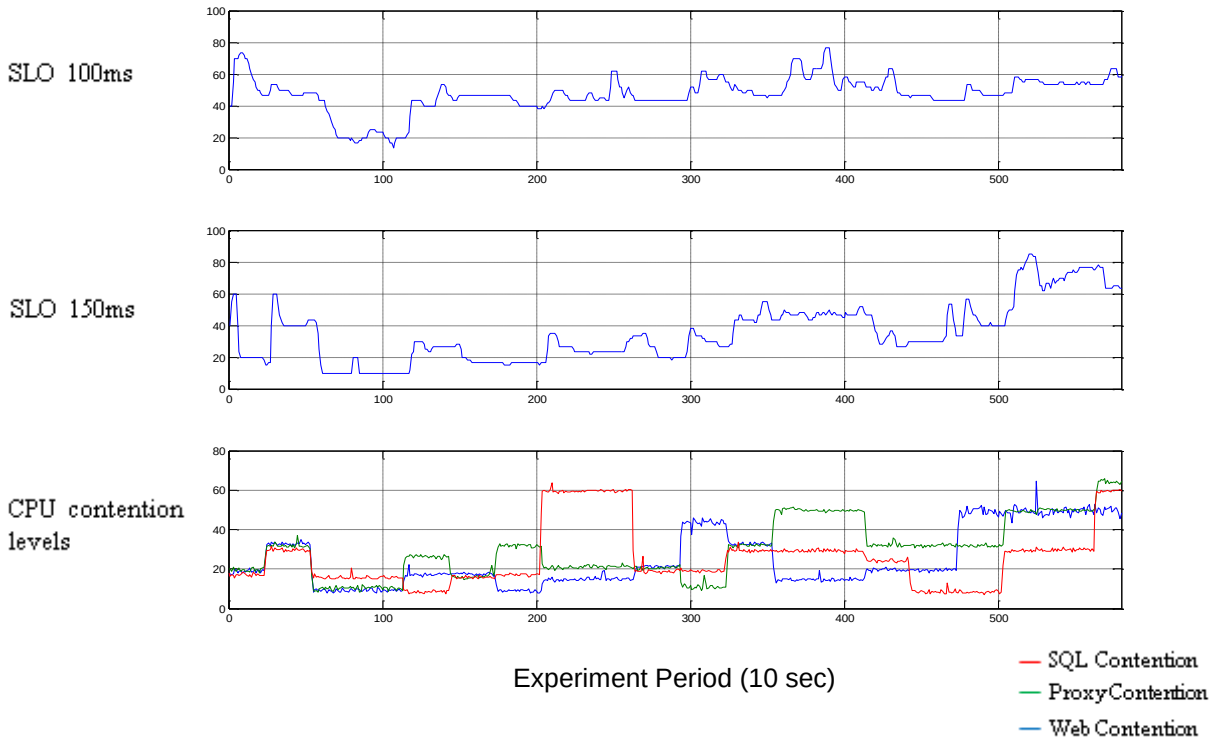


Figure 4-19: Allocated resource levels for dynamic resource allocation test

4.4.3. Change in User Levels

Figure 4-20 shows the TPC-W response time when the number of users is varied during the experiment. We again configure our system to meet either a 100ms or 150ms response time SLO. We also experiment with the VMs resource allocations set statically to either 50% or 10%.

It can be seen from Figure 16 that our system can dynamically adjust resource allocations to meet an SLO despite a varying user level. Our system keeps the response time near the SLO target, whereas the static resource allocation causes response time to vary from 100ms-400ms.

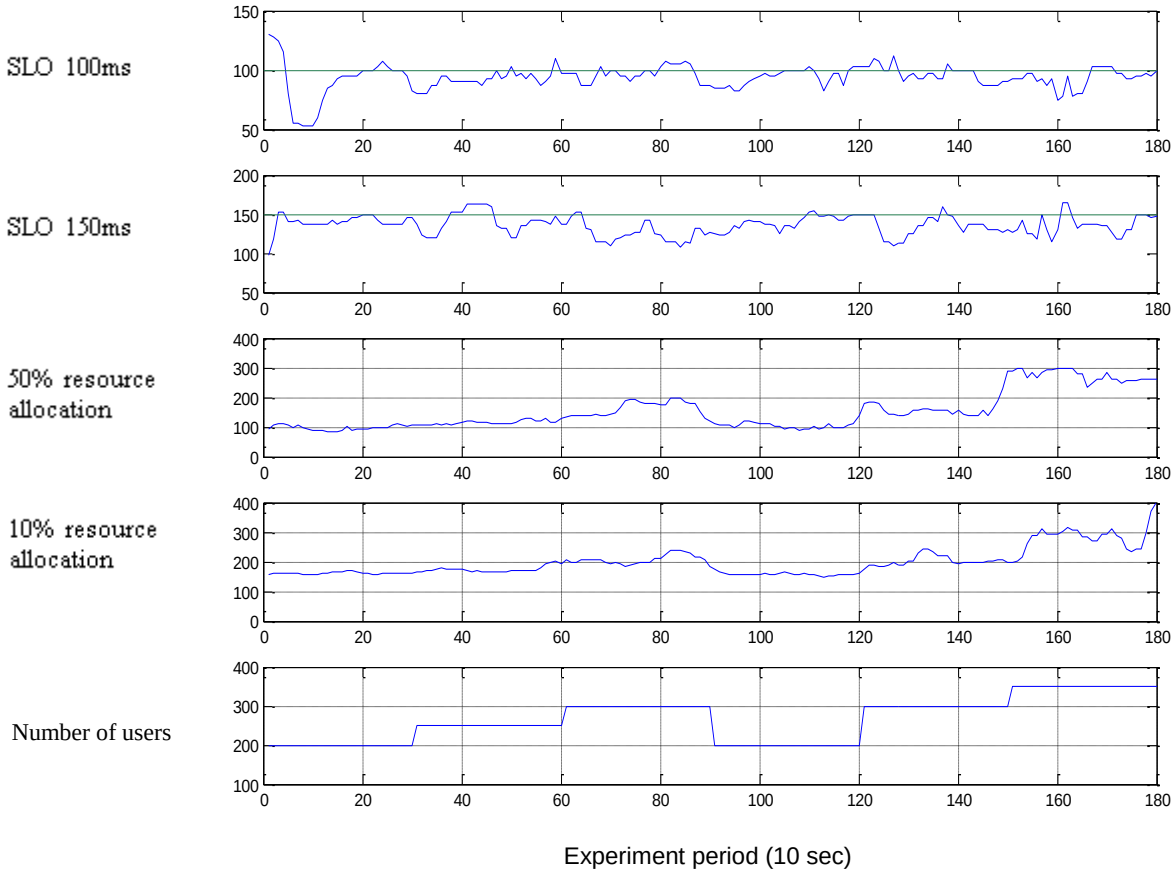


Figure 4-20: Response time results for dynamic and static resource allocations, changing user level

4.5. Conclusion

In this chapter we show that applications comprised of multiple VM tiers can meet SLOs by dynamically allocating host resources. We show that by capturing an application's previous performance, we can model and predict the minimum amount of resources it needs to meet an SLO. Additionally, we show that these models can be interpolated to fit previously unobserved host utilization levels. This allows our dynamic resource control to quickly alter resource allocations when resource utilization levels change.

We evaluate our system using TPC-W and setting response time SLO targets. The host utilization was varied throughout the experiments. Our controller adapts to the changes in host utilization levels, and helps maintain TPC-W's response time within the SLO target. Our controller also assigns the minimum amount of resources required to meet the SLO, allowing the other application running on the same hosts to improve its performance.

5. Input Shaper

In this chapter we present our application workload input shaper. The goal of the Input Shaper is to reduce the variance in VMs' resource utilization levels to ensure their current resource allocations are sufficient to achieve their SLOs. It also limits the amount of resource contention experienced by VMs to further ensure SLO achievement. By reducing the variance in VMs' resource utilization levels we can also reduce the amount of resource overprovisioning required to achieve SLOs. This reduces the total amount of resources required for applications to meet their SLOs.

To achieve the above goals, Input Shaper dispatches requests to one of two datacenter zones: a tightly controlled shaped zone, or a best-effort overflow zone. Requests in the shaped zone are dispatched such that the variances in hosts' resource utilization levels remain low. This is achieved by controlling the type and frequency of requests sent to each VM. The 'pattern' for how to dispatch requests to a VM is created by analyzing the response time and resource consumption of each request type and the amount of resources allocated to the VM. If an incoming request does not conform to any VMs' patterns it is instead dispatched to an overflow zone. VMs in the overflow zone do not have tightly controlled request patterns and therefore may not achieve an application's SLO. Through the integration of application input shaping and dynamic resource allocation, our system allows the achievement of SLOs in situations that cause previously proposed schemes to fail.

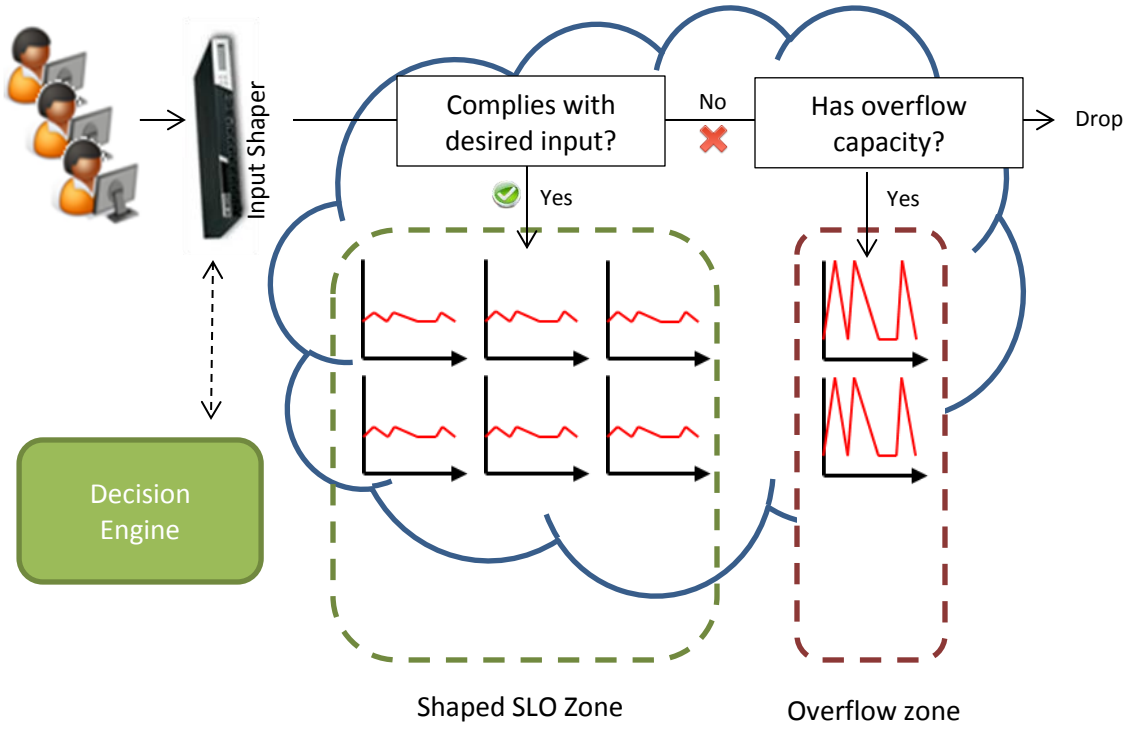


Figure 5-1: Overview of Input Shaper

Figure 5-1 shows an overview of the Input Shaper system. When clients' requests arrive at the Input Shaper, it decides which VM should process each request. When the Input Shaper is initially started, it contains only the information about which VMs are running which applications. At this point it cannot yet make any intelligent shaping decisions, and it balances the incoming requests in a round robin fashion. This means that all VMs and hosts are currently in the overflow zone. The Input Shaper monitors incoming requests' URLs, arrival rates, and response times. Over time it begins using this information to dispatch requests in a more controlled pattern, rather than evenly distributing requests between all VMs.

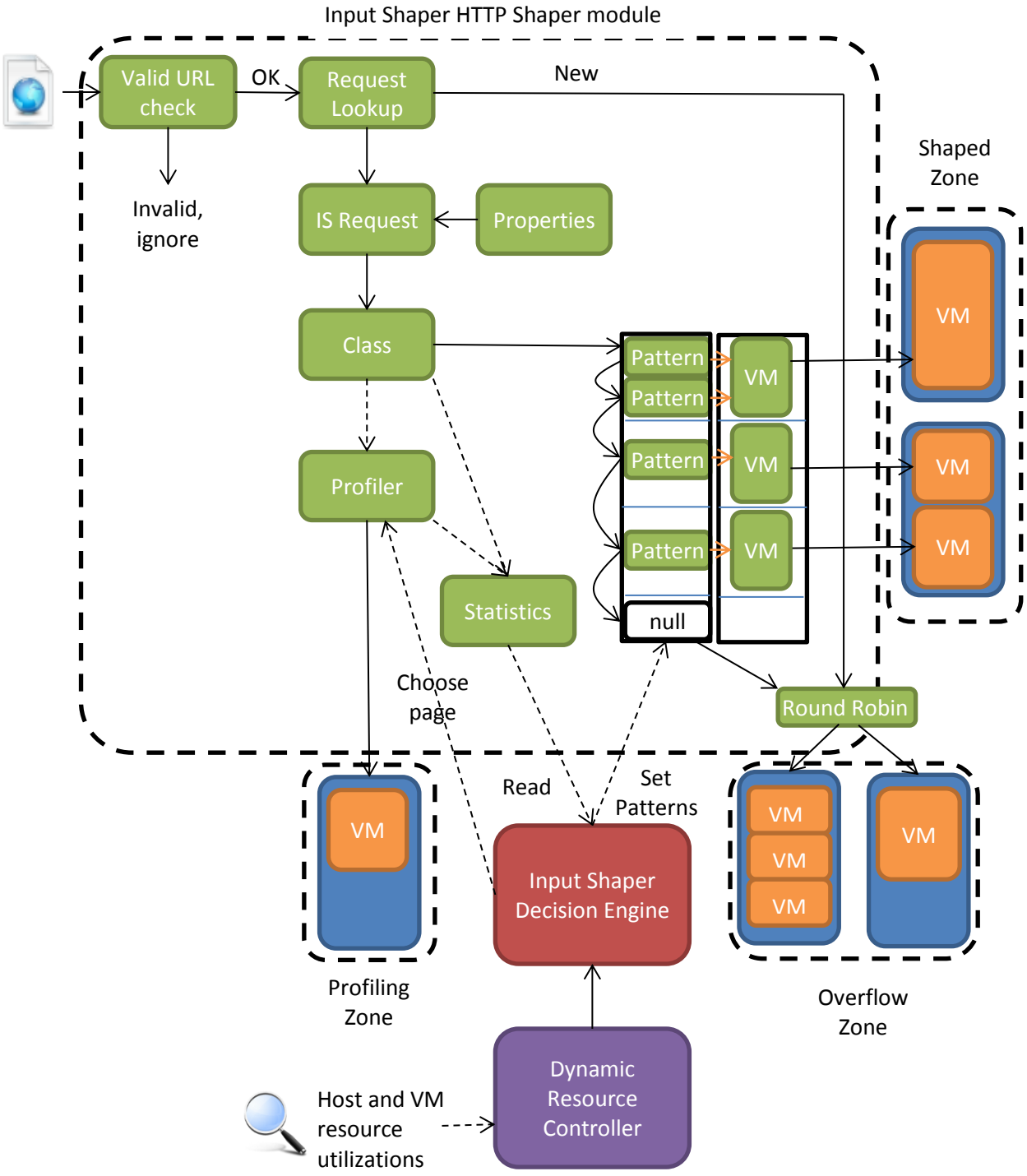
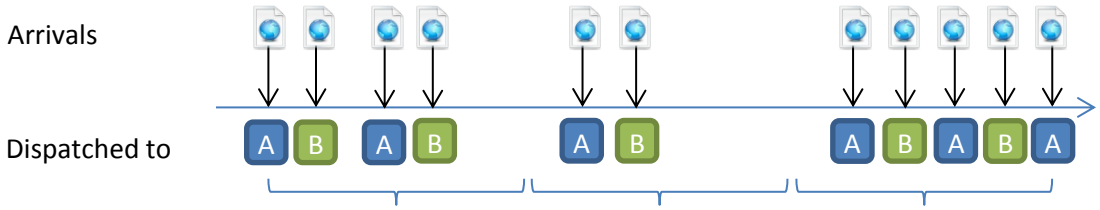
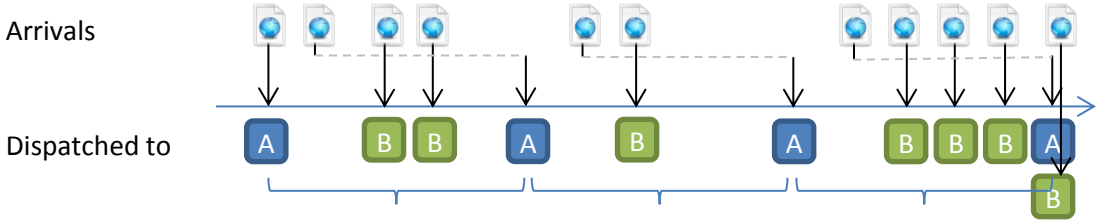


Figure 5-2: Management system overview

Round robin – Average 1.6 requests per period to A



Deterministic dispatch – Average 1 request per period to A



Deterministic dispatch max wait – Average 0.66 requests per period to A

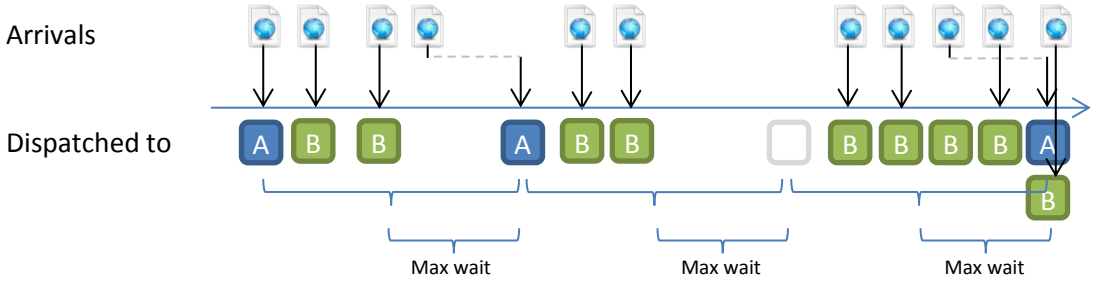


Figure 5-3: Dispatching requests via round robin or rate limited

Figure 5-3 shows incoming requests being dispatched to VMs A and B by either a round robin or deterministic algorithm. When using the round robin algorithm each VM receives an equal number of requests to process. This makes the resource utilization of each VM dependent on the rate of incoming requests to the application as a whole. If the number of incoming requests for the application increases then the resource utilization of each VM also increases. For example, the resource utilization in the third period will be greater than the first or second. This can cause an application to fail to achieve its SLO as it does not have sufficient resources to process all of the incoming requests. It can also cause other applications to fail their SLOs due to increased resource contention. As a result, applications under currently proposed schemes are overprovisioned resources to allow for variations in their workload levels. As the value between VMs' maximum and average workload levels increases, so does the amount of resources they are wasting.

The second scheme shown in Figure 5-3 is a deterministic dispatch algorithm. Rather than dispatching requests to each VM in turn, one VM is dispatched requests to process in a deterministic fashion. This ensures that the number of requests being processed by the VM remains constant over time. If we assume that each request requires the same amount of resources to process, then the resource utilization level of this VM remains constant. The VM can therefore be allocated the correct amount of resources to achieve its SLO and would neither exceed its allocation nor waste resources. If all VMs located on a host are using a similar deterministic scheme they would not be affected by the risk of failing their SLOs due to increased resource contention.

Figure 5-4 shows a simple example where limiting the number of requests processed by a VM can improve an application's SLO achievement rate. By forcing some requests to be processed sequentially, rather than admitting them all to process in parallel, three out of the five requests can achieve their SLO target, rather than zero.

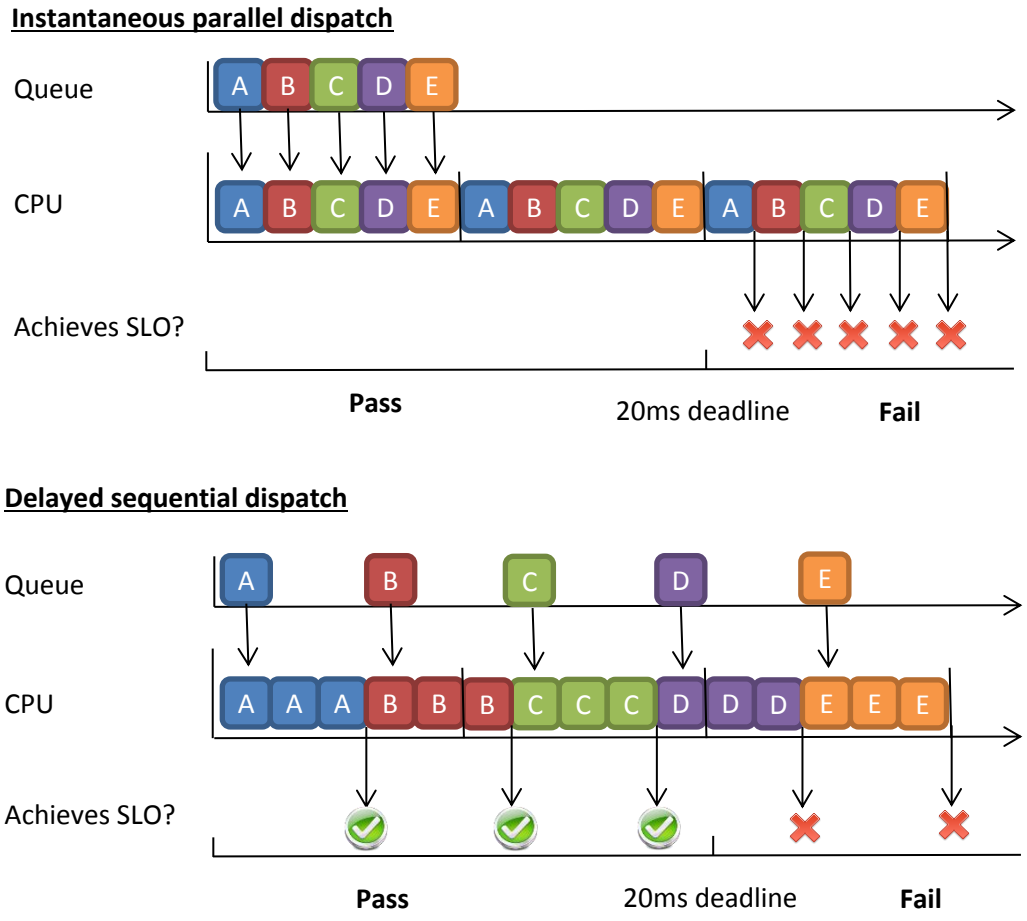


Figure 5-4: Processing fewer requests can achieve higher SLO achievement

While deterministic request arrival rates and processing times would be ideal, a real world application is unlikely to exhibit this behavior. Also, as shown in Figure 5-4, requests cannot be delayed indefinitely just to suit a VM's desired request pattern as they would fail to achieve their SLOs due to the wait time. To prevent this Input Shaper uses a maximum delay estimate for each request based upon its observed response time and the specified SLO. For example, if we expect a request to take 30ms to process on a VM and the SLO has a maximum response time of 100ms, the request could be delayed up to 70ms. However, as shown in the last scheme of Figure 5-3, this can lead to having no suitable requests available to send to a shaped host. We address this in Section 5.1.3 by relaxing the request dispatch interval and potentially dispatching requests slightly early or late depending on the VMs' current estimated resource utilizations.

To calculate requests' maximum delays we use the response time at a percentile value of a request's response time CDF. This allows any percentage of requests to achieve satisfactory service, rather than only the 'average' request that many current schemes satisfy. Figure 5-5 shows the response time CDF for the C-MART search page as monitored by Input Shaper. If our SLO was 150ms and we want 80% of the search results returned within this time a search request's maximum wait time would be:

$$\begin{aligned} \max delay &= SLO_{RT} - \%tile\ RT\ CDF \\ &= 250ms - 160ms \\ &= 90ms \end{aligned}$$

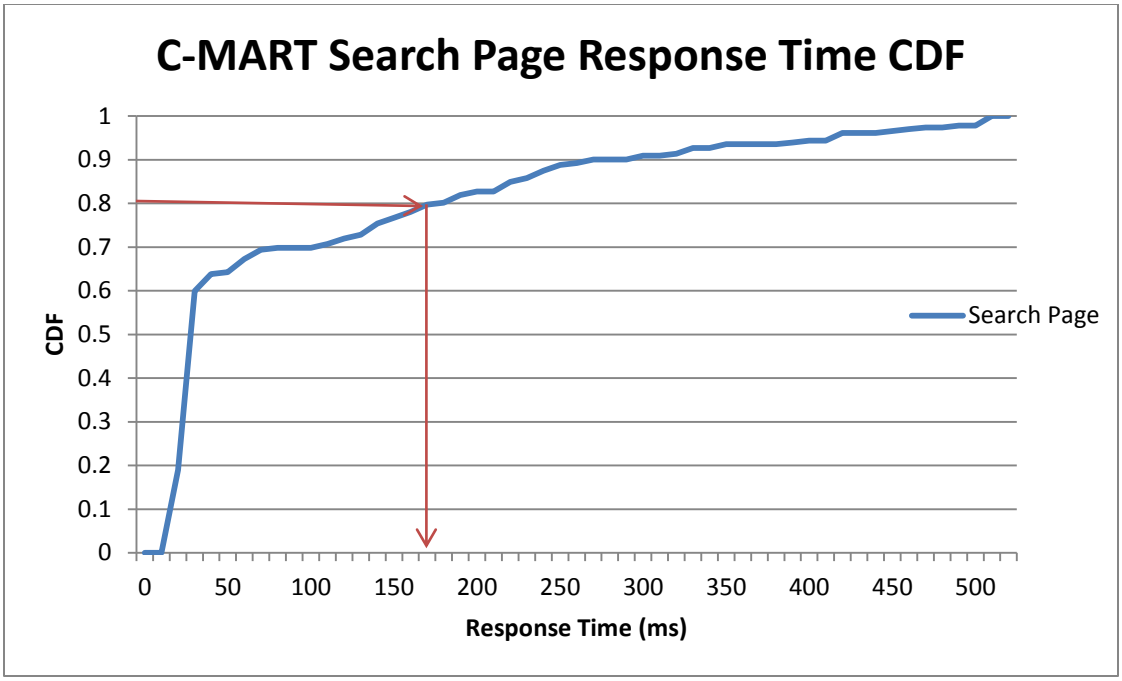


Figure 5-5: Response time CDF of C-MART search page

In addition to requests having different maximum wait times, different requests are also likely to consume different amounts of hosts' resources while processing. Even when using the deterministic dispatch scheme in Figure 5-3, if requests are utilizing different amounts of resources to process then VMs' resource utilizations will still vary. Real world applications' requests can consume a different order of magnitude of resources to process. Therefore, Input Shaper also considers requests' expected resource utilizations when making its VM selection decision. The Input Shaper runs as an HTTP request balancer and only knows the address of the VMs it is dispatching requests to. It does not have direct access to hosts' or VMs' resource utilization levels. Instead, the Input Shaper's external Decision Engine works with our Dynamic Resource Controller to collect these metrics. To identify the resources consumed by each request type, the Input Shaper performs a profiling stage. During request profiling it directs only

certain requests to a chosen VM so that the response time and resource utilization of those requests can be observed in isolation. Further details on the profiling stage are given in Section 5.1.4. After requests' estimated resource utilizations are known, the Input Shaper uses this information to manipulate VMs' resource utilizations via the request types they are sent.

For larger applications, Input Shaper dispatches requests to multiple VMs simultaneously. It attempts to utilize each VM's resource allocations as much as possible to reduce resource waste. For each incoming request, it iterates through its array of shaped VMs and compares the request to each VM's desired pattern. The request is assigned to be dispatched to the first VM whose pattern it matches. This results in the behavior of the first VM being 'filled', or satisfied, first, with VMs later in the array being less likely to fully utilize their resource allocations. For example, an incoming request waits for the first VM that would satisfy its SLO rather than being instantly dispatched to a VM later in the array. We choose this approach as we are not attempting to balance load equally, but instead shape the resource utilization of some VMs with as low a variance as possible. If the last VM in the shaped array is wasting many resources then it should instead be reallocated to the overflow zone, or removed entirely if it is no longer required. This is discussed further in Section 5.1.3.

In the remainder of this chapter we discuss the design, implementation and experimental results of Input Shaper. We show that by shaping applications' workloads we can reduce resource waste on shaped hosts by as much as 75%. Reduced waste results in reduced hardware requirements. We show up to a 45% reduction in resources requirements to host all of a cloud's applications.

5.1. Design

The Input Shaper is designed to work with generic web applications in cloud computing environments. As it needs to dispatch requests to the VM of its choosing, VMs must process requests in a stateless fashion. This is already a common design pattern for cloud based applications as it allows applications to easily scale up and scale down [81]. The complete Input Shaper system is split into two components. The first is an HTTP shaper module that decides which VM each individual request is dispatched to. The second is a Decision Engine that analyses requests' response times and resource utilizations to create the general request patterns that the HTTP shaper module follows. Input Shaper is segregated like this to reduce the load on the HTTP shaper as we want to minimize the processing time experienced by each request at the Input Shaper.

As the Input Shaper is integrated with our Dynamic Resource Controller, there are two ways in which Input Shaper can ensure applications achieve their SLOs. Firstly, the Input Shaper can limit the number of requests sent to VMs to only those that can be processed within their SLOs. If the Input Shaper is under or over utilizing VMs' resources, then the VMs' request dispatch pattern can be altered. This results in hosts' resource utilizations being maximized. This does not necessarily mean that hosts will be consuming all of their available resources. For example, to achieve a given SLO's response time we may need to keep a host's CPU utilization under 70%. In this case utilizing 70% of the CPU cycles equal full utilization.

The second way that Input Shaper can be used to achieve SLOs is to keep a VM's request dispatch pattern constant and allow the Dynamic Resource Controller to alter the VM's resource

allocations. Using Input Shaper and Dynamic Resource Controller together is preferable to using Dynamic Resource Controller alone as it reduces the amount of overprovisioning required to achieve SLOs. As the variation in VMs' resource utilizations decreases so does the required amount of resource overprovisioning.

The following section describes the main components of the Input Shaper system

1. **Admission Control** which removes the potential for hosts to become overloaded and cause SLO failures
2. **Generating Request Patterns** which tells the HTTP shaper module how it should dispatch requests to achieve applications' SLOs
3. **Request Shaping** which is the process by which the HTTP shaper decides which individual requests are sent to which individual VMs
4. **Profiling** which allows Input Shaper to gather information about requests to further reduce variance in VMs' resource utilization levels

5.1.1. Admission Control

In Chapter 4 we show that applications can achieve response time SLO through dynamic resource allocation. However, this is dependent on sufficient resources being available to allocate to each application. If all applications experience a sudden rise in workload levels, hosts will become overloaded resulting in performance degradations. Cloud computing environments can provision additional VMs on other hosts to increase applications' available resources, but this takes multiple minutes to complete. Williams et al. [28] show that up to 88% of transient

workload overloads last less than two minutes. By the time a new VM is provisioned it may already not be needed.

To overcome this, current schemes constantly overprovision applications to achieve their SLOs. However, in reality, even this would not be enough. By balancing load between applications' VMs, we place all of those VMs at risk of overload. In addition, this potentially puts all other customers sharing the same hosts at risk of overload due to increased resource contention. Flash crowd events can cause applications' workload levels to change by multiple orders of magnitude within minutes [69]. Therefore, schemes such as [37], [38], [39], and [40] are unlikely to work outside of the research environment.

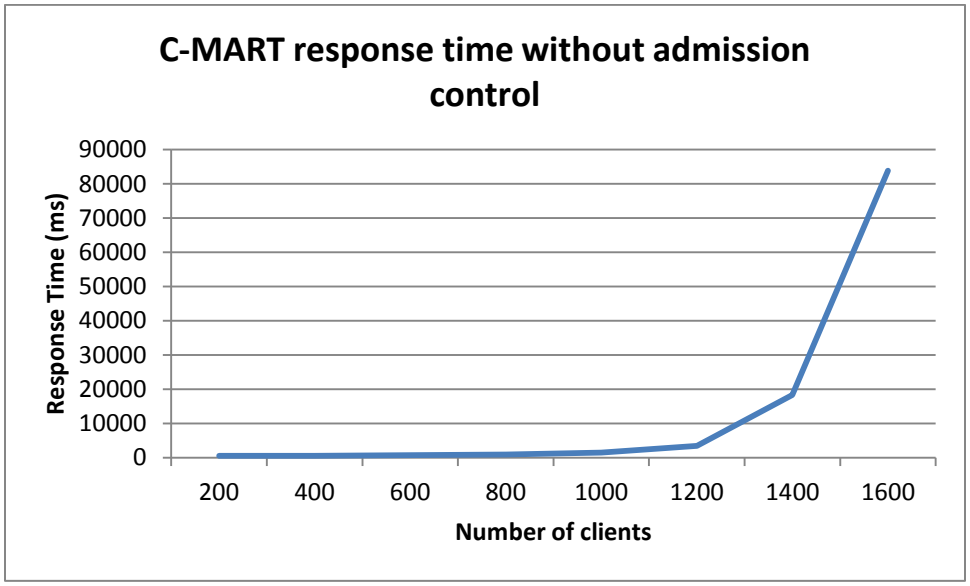


Figure 5-6: The effect of not having admission control

Figure 5-6 shows the response time of C-MART as the number of clients is increased. In this case C-MART is running on its own dedicated hosts, so it is receiving the maximum amount of resources available. It can be seen that without admission control the application can easily

fail to achieve its SLO. Once the host's resources are fully utilized the application's performance becomes exponentially worse.

To address the above problems the Input Shaper performs admission control to prevent too many requests being admitted to applications. VMs in the shaped zone have their own form of admission control as they are only dispatched requests if they conform to the VM's pattern. However, in the overflow zone requests are dispatched in a round robin fashion. Although the overflow zone does not guarantee requests' response times, it is still undesirable to completely overload VMs there. The number of requests dispatched to VMs in the overflow zone is set by the Decision Engine. For each application, it estimates the number of requests that should be admitted by dividing the resources available to the application in the overflow zone by the average resource utilization of the application's requests.

$$u_{app} = \frac{\sum requests\ admitted}{\sum VMs' resource\ utilizations}$$

$$r_{app} = \sum VMs'\ hosts'\ maximum\ resources$$

$$Max\ admissions\ per\ second\ = \frac{r_{app}}{u_{app}}$$

As multiple applications may be admitting their maximum load, this does not prevent the overflow zone's resources from becoming depleted. It does however restrict the upper limit on the amount of overload. This is the same principle used by [35] to overbook resources, and it is the nature of statistical multiplexing. While hosts will sometimes become overloaded, [35] shows that the increase in resource utility can be as high as 500%. This scheme is a good fit as

the overflow zone is designed to process load variations and not necessarily achieve the SLO targets.

5.1.2. Request Patterns

Request patterns tell Input Shaper's HTTP module what types of requests, and when, to dispatch to each VM. The request patterns are decided by the Decision Engine and then uploaded to the HTTP shaper module. The HTTP shaper module attempts to follow the patterns as accurately as possible to ensure VMs receive the workloads they are provisioned for. The request shaping mechanism is discussed further in Section 5.1.3.

The creation and following of patterns is intentionally separated to reduce the processing burden on the HTTP shaper module. As every incoming request is processed by the HTTP shaper module we want to keep the processing time low. Additional processing time at the HTTP shaper module only increases the likelihood that requests fail to achieve their SLOs. In our current implementation the processing time of each request is on the order of 0.1ms; this is negligible compared to the time spent at other tiers of the applications.

To reduce resource overprovisioning it is desirable that VMs' resource requirements remain static over time. A completely static workload level would allow the correct resource allocation level to be set only once, after which there would be no resource waste and no SLO violations. To approach this state, we give VMs deterministic request arrival rates. However, as the resources consumed by requests differ, a deterministic request arrival rate alone is insufficient.

Symbol	Description
SLO_{RT}	The response time target specified in the SLO
$SLO_{\%}$	The percentage of requests that should achieve the SLO target
$VM^{i}_{ResourceAllocation}$	The amount of resources that are allocated to the VM on its host
$VM^{i}_{Contention}$	The amount of contention that will be experienced by the VM. This is the sum of the resource utilizations from the other shaped VMs located on the same shared host, i.e. $\sum_{j\neq i}^{n} VM^{TargetUtil}_{ResourceAllocation}$
$VM^{i}_{TargetUtil}$	The VMs' maximum expected resource utilization where its response time is still less than its SLO
$Request^{j}_{ArrivalRate}$	The inter-arrival rate CDF for a given request type of type j
$Request^{j}_{AverageUtil}$	The average resources consumed while process request type j
$Request^{j}_{ResponseTime}$	The response time CDF for the request type j
$Request^{j}_{MaxDelay}$	The maximum amount of time a request can be delayed based on its response time CDF and the SLO
$Pattern_{rps}$	The number of requests per second to admit under the pattern

Table 5-1: Data used during pattern creation

The first step in calculating a VM's request pattern is to calculate the VM's desired resource utilization level. Its desired resource utilization level depends on the amount of resources it is allocated on its host. In Chapter 4 we discuss using applications' performance models to calculate their resource requirements for a given workload level. The same data can instead be used to estimate the desired workload level for a given resource allocation. For

example, if a host contains three VMs, we could allocate each of them 1/3 of the hosts' resources and calculate the maximum workload for each that still allows them to achieve their SLOs. When using the Input Shaper it is preferable to shape the request rates of all VMs on a host. Otherwise, changes in resource contention levels can still affect VMs' performances and cause SLO violations. By shaping all VMs on a host the amount of resource contention is limited and also exhibits low variance.

The next step is to calculate the number of requests per second that are needed to achieve the resource utilization level, $VM^{i}_{TargetUtil}$. To calculate this we could simply divide the desired resource utilization by the average resource consumption of all incoming request types:

$$\frac{VM^{i}_{TargetUtil}}{\sum_{j=0}^{n} Request^{j}_{AverageUtil}} = requests\ per\ second$$

As previously mentioned, requests' resource consumptions can differ by an order of magnitude or more. While the above equation would average VMs' resource consumptions over the long term, short term overloads of resources would cause SLO violations. To mitigate this we limit the types of requests that match to each VM's request pattern based on $Request^{j}_{AverageUtil}$ and $Request^{j}_{MaxDelay}$. For example, we may only match requests that consume the same amount of resources within 20%. This prevents requests that consume orders of magnitude different amounts of resources being matched by the same request pattern. To achieve this, requests are classified. The information used to classify requests is acquired via profiling, as described in Section 5.1.4.

A request class is a set of requests that can be considered as a single type by the HTTP shaper module. A request class is typically composed of requests that consume similar amounts of resources to process. When a request enters the Input Shaper it is checked to see if it belongs to a request class. If it does belong, the HTTP shaper uses that class's properties to make its dispatch decision. By default, requests belong to the default request class that is always dispatched to the overflow zone. This prevents request types that we have not previously observed from being admitted into the shaped area of the datacenter where they could disrupt applications' SLOs.

To decide which request class a pattern should admit to meet a VM's $VM^{i}_{TargetUtil}$ each class is ranked for suitability. Classes are first filtered on the criterion that they must have sufficient incoming requests to fulfill the VM's target utilization. For example, if a class is only receiving three requests per second but it would take 100 of those requests to meet the VM's target utilization, then that class is not a suitable candidate. When performing this calculation we must consider the number of requests from each class that are already being matched by other VMs' patterns. The greater the number of requests that are not currently matched the greater the probability that there will be a sufficient number of requests to satisfy a new request pattern.

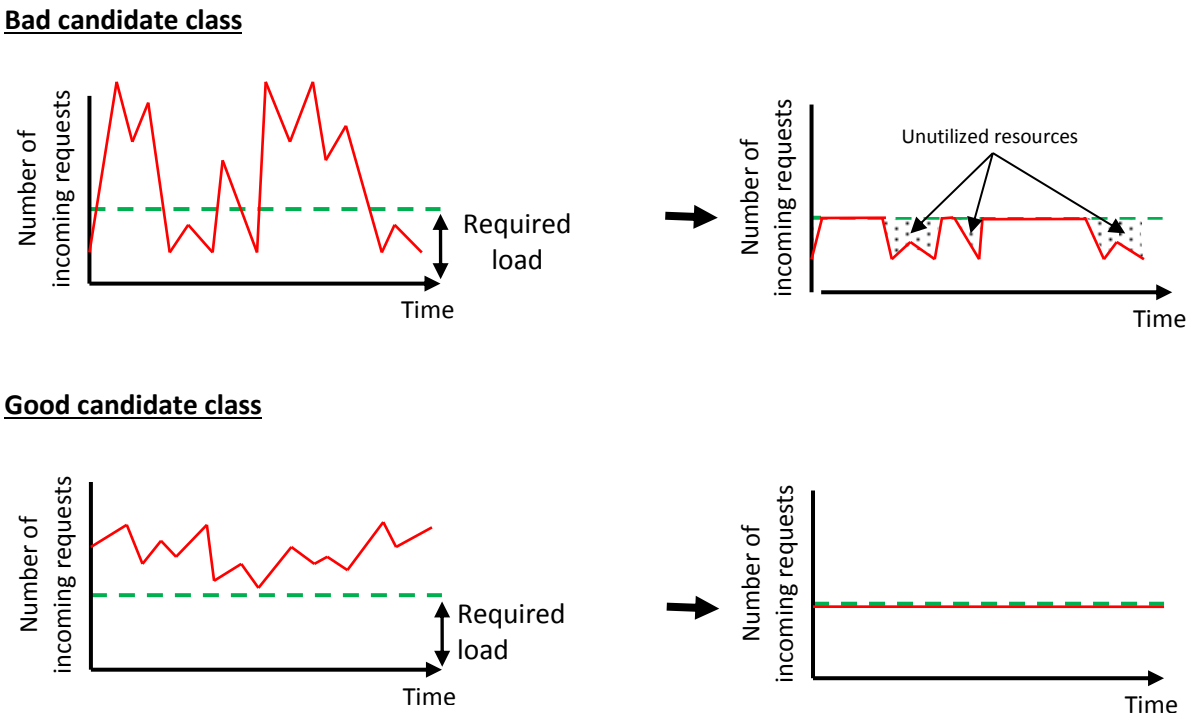


Figure 5-7: Good and bad candidates for shaping

The second criterion considered is that the inter-arrival time of the requests must be suitable to achieve $VM^{i}_{TargetUtil}$, as illustrated in Figure 5-7. Even if requests' aggregate resource consumption is great enough to satisfy $VM^{i}_{TargetUtil}$ they may arrive such that they cannot distribute that consumption over time. For example, if requests arrive in a highly bursty pattern, all of their resource consumption occurs at the same time. To decide if a class is suitable for a pattern we must consider the average $Request^{j}_{MaxDelay}$ for the class. If requests arrive in a bursty pattern and we cannot delay the requests without violating the VM's SLO, then we cannot successfully reshape the requests to fulfill the VM's request pattern. However, if

requests arrive in a bursty pattern and their max delay is large we can reshape them to the VM's requirements without violating the SLO.

As an example, let us assume that we have the following SLO: 85% of requests respond within 150ms. We will use C-MART's View Item page as a candidate match to the pattern. We will assume that to fulfill the VM's desired 20% resource utilization level five View Item pages per second must be dispatched to the VM. From the response time CDF of the View Item page in Figure 5-8 we can calculate that 85% of View Item requests complete within 20ms. Therefore, to achieve the SLO each request has a maximum delay time of 150ms-20ms = 130ms.

Using the View Item request's inter-arrival CDF in Figure 5-9, we can calculate that given a maximum 130ms maximum delay time there is a 95% probability that the View Item page will match the pattern during each dispatch period. Naïvely we can calculate the expected amount of wasted resources as (1 – 0.95) *100 = 5%. This waste occurs when there is not a request to dispatch, so the VM's resource allocation remains unutilized. However, if there is not a request to dispatch at the desired deterministic time the next View Item to arrive can be dispatched instantly as the VM's resources are available. Rather than the resources remaining idle for an entire dispatch period, they are only idle until the next request arrives. This changes the expected amount of resource waste to (using data from Figure 5-9):

$$original\ waste\ estimate * expected\ idle\ time$$

$$\left((1 - 0.95) * 100\right) * \left(\frac{15}{200}\right) = 0.375\%$$

As the expected amount of resource waste is low, the View Item request is a good candidate to fulfill this pattern. The amount of acceptable resource waste is a parameter that would be tuned by an administrator, or could be automatically calculated as: any value less wasteful than the current waste in the overflow zone. More generally the expected waste is calculated as:

$$Pattern_{rps} = \frac{VM^{i}_{TargetUtil}}{Request^{j}_{AverageUtil}}$$

$$Request^{j}_{MaxDelay} = SLO_{RT} - Request^{j}_{ResponseTime}{}^{-1}(SLO_{\%})$$

$$Request^{j}_{Arrival} = Request^{j}_{ResponseTime}(Request^{j}_{MaxDelay})$$

$$Request^{j}_{NoArrival} = 1 - Request^{j}_{Arrival}$$

$$E(arrival) = Request^{j}_{ResponseTime}{}^{-1}(\frac{100 - SLO_{\%}}{2})$$

$$E(Resource\ Waste) = Request^{j}_{NoArrival} * 100 * E(arrival)$$

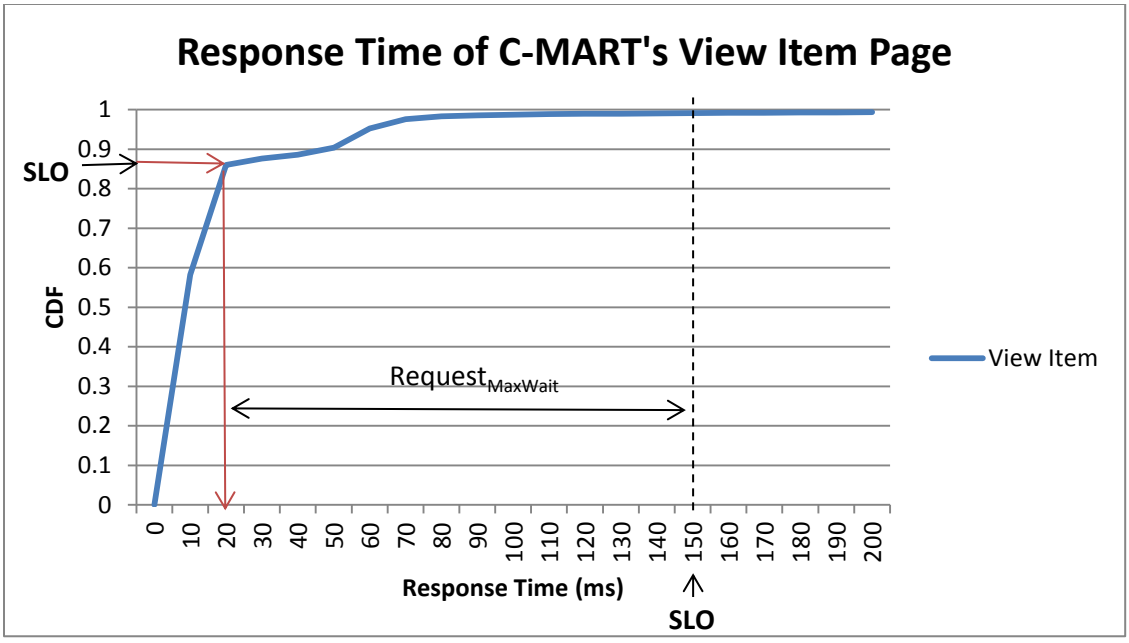


Figure 5-8: Calculating max delay time

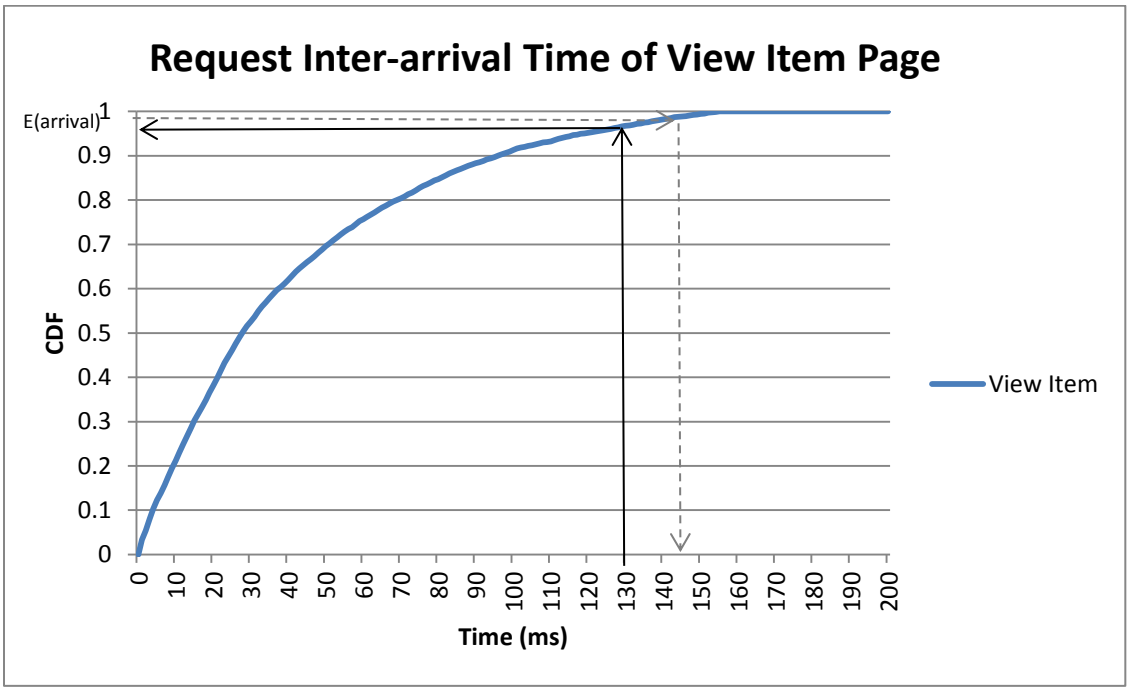


Figure 5-9: Calculating probability the request matches pattern

If a suitable candidate class is found to create a shaped pattern, the pattern uploaded to the HTTP shaper module is:

Pattern id: The identity of the pattern

Class id: The class that the pattern accepts

VM id: The VM that conforming requests are dispatched to

Requests per second: The number of requests per second to dispatch

VM resource utilization target: The desired resource consumption of the VM

The VM resource utilization target is specified as part of the pattern because the HTTP shaper module does not perform analysis on the data it collects. It does **not** identify each request in a class and calculate the VM's resource utilization target based on the requests per second and average resource utilizations. Instead it only does what it is told by the Decision Engine. The VM resource utilization target is used by the shaper algorithm to achieve VMs' desired targets as accurately as possible, as described in Section 5.1.3.

In addition to specifying a single class to satisfy a VM's target utilization level, multiple classes can be specified in multiple patterns to achieve the target. For example, if a VM wanted to achieve a 50% resource utilization rate it could receive all resource consumption from one class, or half each from two different classes. Any number of classes and patterns can be combined to achieve VMs' resource utilization targets. Each pattern is evaluated independently to prevent the potentially large differences in requests' resource consumption levels causing short term resource overloads.

In the following section we discuss how the requests are actually shaped and dispatched from the HTTP shaper module.

5.1.3. Request Shaping

The ability to control VMs' incoming request rates is where Input Shaper achieves most of its benefits. It allows Input Shaper to make use of resources more efficiently compared to current resource allocation schemes. The first major benefit is the ability to reshape VMs' total resource utilization levels. This allows hosts to be packed with VMs more densely as VMs' resource utilizations are shaped to fit hosts' available resources, as shown in Figure 5-10. The combined Input Shaper and Dynamic Resource Controller are being proactive with regards to VMs' resource utilization levels, rather than only reactive. This is in contrast to current schemes, such as [30], [31], [32] and [33], that instead choose the number of active hosts in response to VMs' resource utilizations.

The second benefit to shaping requests is the ability to reduce the variance in VMs' resource utilization levels. This reduces the amount of resource overprovisioning required to ensure VMs achieve their SLOs. Figure 5-11 demonstrates a simple example where shaping a VMs' incoming request rate reduces the variance in its resource utilization. In this experiment a simple web request that performs CPU calculations is dispatched at either a random or deterministic rate. It can clearly be seen that dispatching the requests at a deterministic rate greatly reduces the variance in the VMs' CPU utilization. The variations in the VMs' CPU utilization when receiving the shaped requests is consistent with the +/- 2% CPU utilization that we observe in VMs due to housekeeping functions.

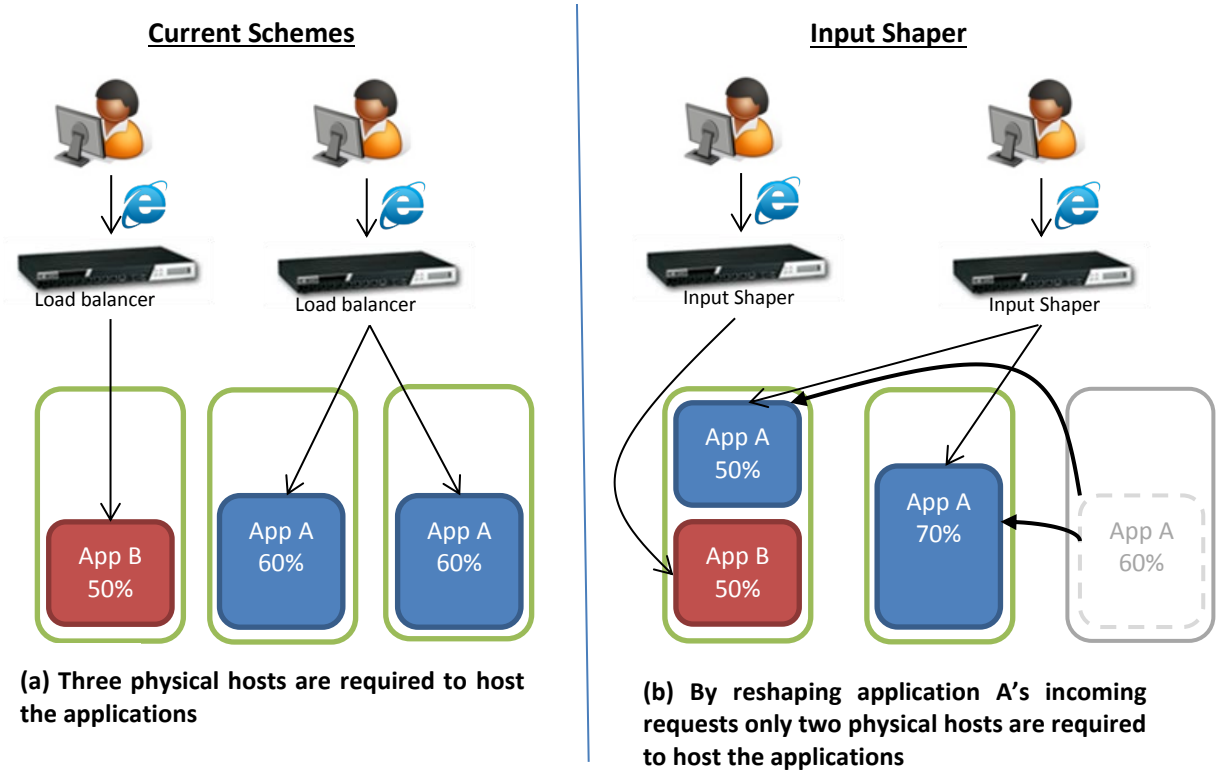


Figure 5-10: Example of reshaping VMs' resource utilizations to reduce number of servers required to host them

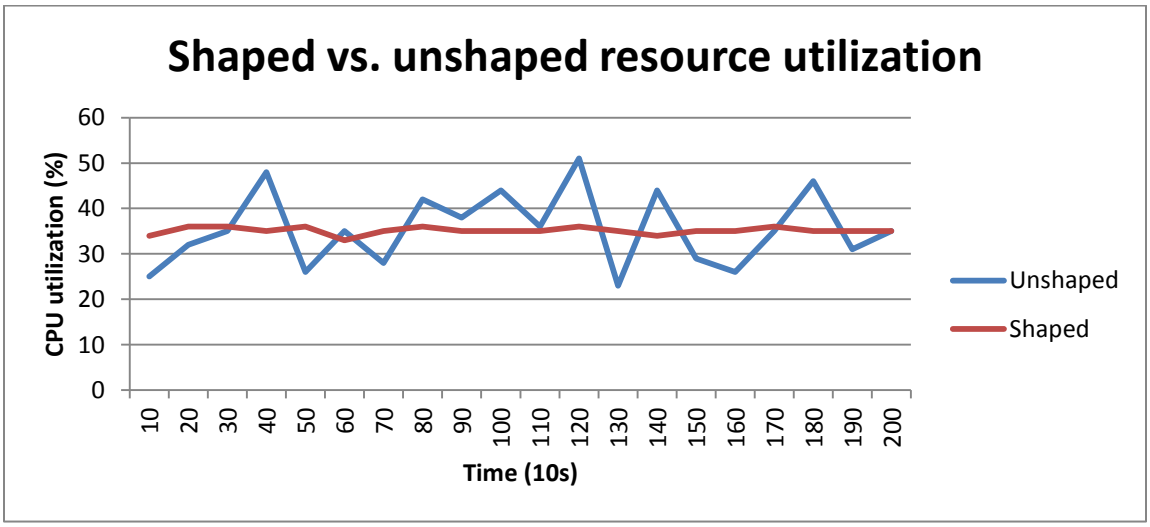


Figure 5-11: Shaping a simple request to reduce variance in a VM's CPU utilization level

When a request arrives at the HTTP shaper module its URL is checked against the list of URLs that Input Shaper is accepting. If the request belongs to an application that is being shaped the request's properties are retrieved using its URLs at the request ID. If the request type has not been observed before a new property's entry is created with default values. By default a new request type will always be sent directly to the overflow zone. If the request has additional properties set by the profiler those will be used instead of the default values. If the request has a class assigned other than the default class it will be compared to VMs' configured request patterns. If it conforms to any of the patterns it will be dispatched to that VM, otherwise it is dispatched to the overflow zone. This process is shown in Figure 5-12.

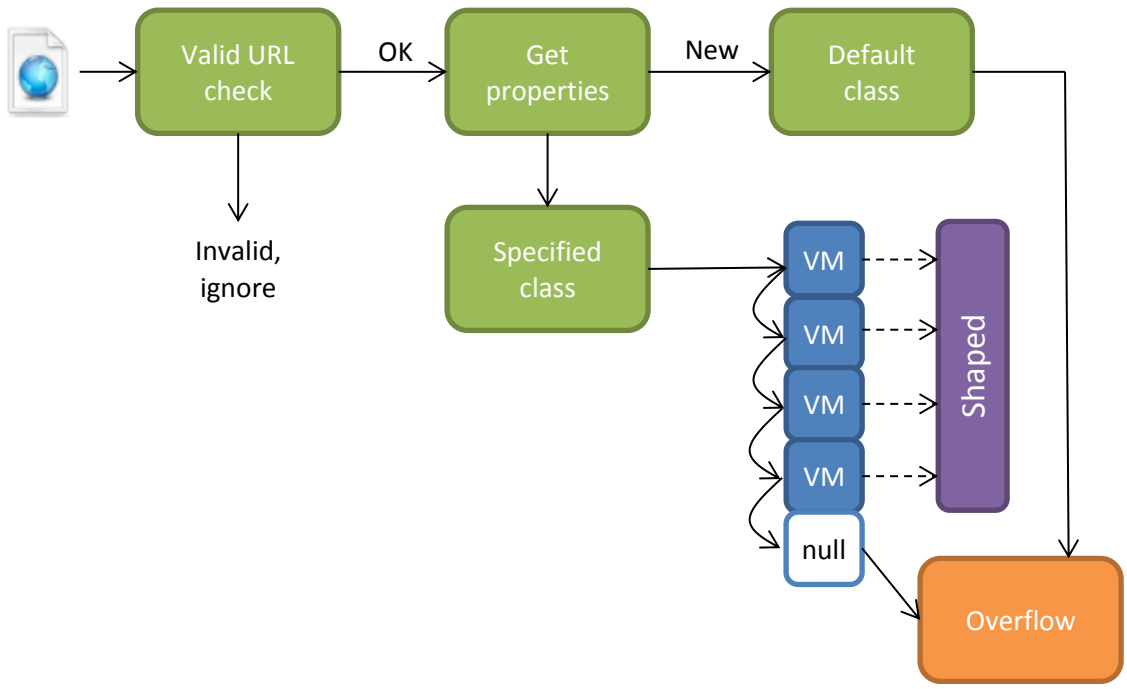


Figure 5-12: Classifying incoming requests and comparing to VMs' patterns

The current dispatch algorithms that Input Shaper supports are round robin and deterministic dispatch. The round robin scheme is used to distribute traffic in the overflow zone. The deterministic algorithm is used for VMs in the shaped zone. The parameters needed for the shaping algorithm are provided by the Decision Engine. In the following examples we assume that the incoming requests all belong to the same class. The first step in dispatching requests deterministically is to only dispatch requests with a *dispatch_interval* = $\frac{1000ms}{Pattern_{rps}}$, as shown in Figure 5-13.

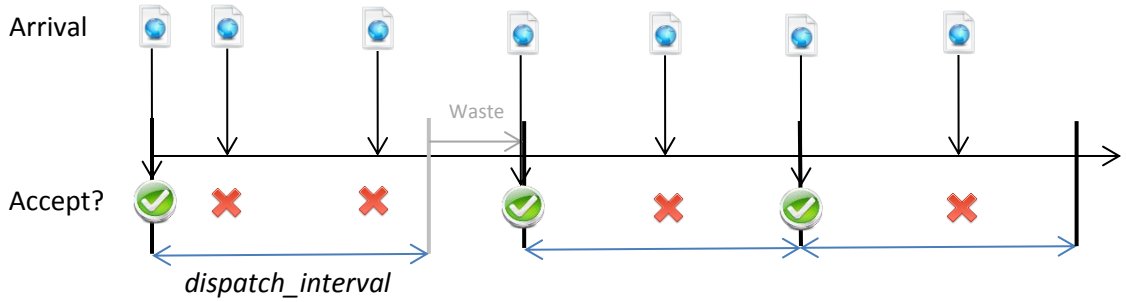


Figure 5-13: Minimum delay between request dispatches

The scheme in Figure 5-13 prevents VMs from exceeding their resource allocations as it rate limits the number of requests dispatched to the VM. The HTTP shaper module maintains a variable *last_dispatch_time* for each pattern to remember when it last sent a VM a request. When a new request arrives it conforms to a pattern if:

$$current\_time \geq last\_dispatch\_time + dispatch\_interval$$

It can be seen in Figure 5-13 that there is a period where no requests were dispatched to the VM even though the VM has resources available to process it. To address this we can

delay the dispatch of requests so that there are requests available to dispatch when we need them, as shown in Figure 5-14.

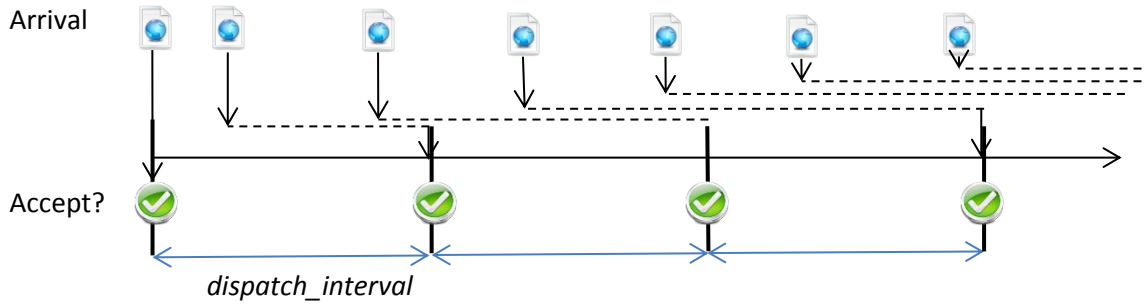


Figure 5-14: Delay requests to reduce resource waste

To achieve this, Input Shaper maintains a variable *next_available_dispatch*. When a request is admitted by the pattern, *next_available_dispatch is* increases by *dispatch_interval*. If we initially start with *next_available_dispatch=current_time, dispatch_interval=10* the values for the first three requests in Figure 5-14 are:

next_available_dispatch = next_available_dispatch + dispatch_interval;

Arrival	*next_available_dispatch before*	*next_available_dispatch after*	*dispatch_interval*
Request 1	0	10	10
Request 2	10	20	10
Request 3	20	30	10

Table 5-2: Requests' next available dispatch time

Although this prevents a VM from underutilizing its resource allocation, delaying requests by long periods will cause them to violate their SLOs. To address this, the *max_delay* parameter is used to decide if a request should be accepted by a pattern and be queued for dispatch, as shown in Figure 5-15.

```
If(next_available_dispatch <= current_time + max_delay){
        if(next_available_dispatch<current_time){
                next_available_dispatch = current_time + dispatch_interval;
                dispatch request now();
        }
        else{
                dispatch request in (next_available_dispatch - current_time);
                next_available_dispatch = next_available_dispatch + dispatch_interval;
        }
}
else reject request;
```

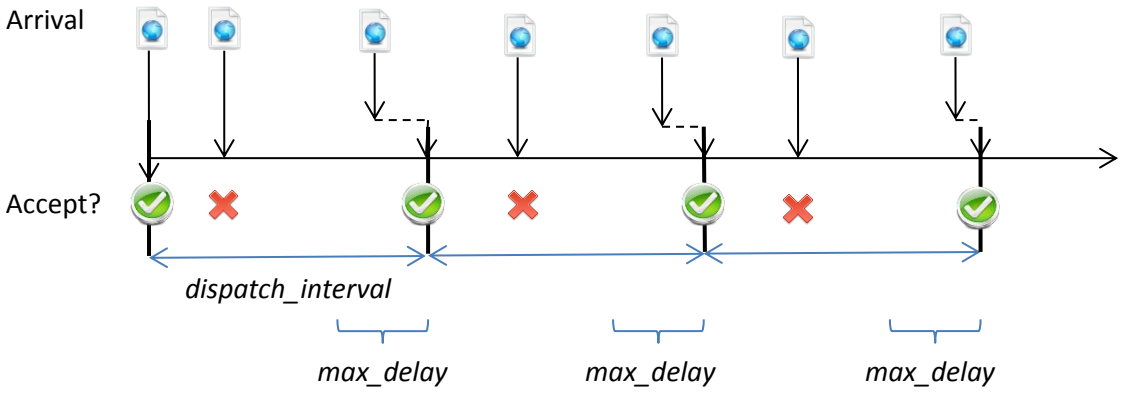


Figure 5-15: Using max delay parameter to prevent SLO violations

The max_delay parameter prevents too many requests from exceeding their SLO's response time due to additional queuing delay. Patterns set by the Decision Engine are designed to maintain VMs' resource utilization levels at a deterministic level. This is achieved by sending additional work to the VM at the same rate that it completes it. Over the long term using requests' average resource consumptions is sufficient to obtain the correct target resource utilization level. However, over the short term these variations could cause additional SLO violations. For example, if we dispatch multiple requests that consume more than the average

resources in multiple consecutive periods, then the hosts may very temporarily become overloaded.

To address this problem we relax the exact nature of the dispatch decision to allow some variability, as shown in Figure 5-16. If we find that we are dispatching multiple larger than average requests we will increase the next dispatch time slightly, and visa-versa if we are dispatching multiple less than average requests. As the goal is to reduce short-term resource utilization variance, we clear the error every 100ms to prevent large positive or negative values accumulating. Although a large negative or positive value may indicate poor request classification or pattern choice, it is not the HTTP shaper's job to make this decision. It instead only reports its error values and maintains the average resource utilization as best it can. The amount of error for patterns can be analyzed by the Decision Engine to help decide when patterns should be altered or removed.

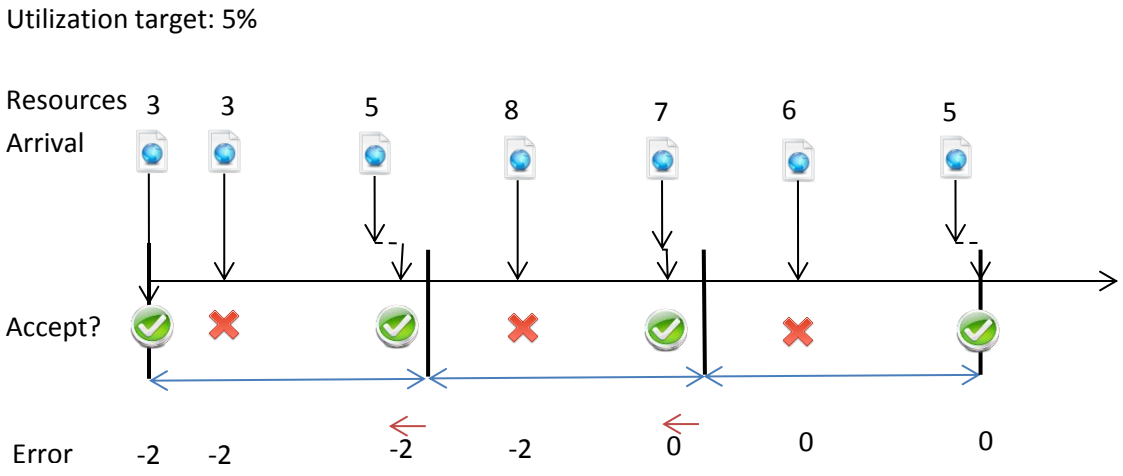


Figure 5-16: Relaxing the deterministic pattern

The amount of time that a request's actual dispatch varies from the deterministic time depends on the relative size of the error. The greater the error term the greater the deviation from the deterministic dispatch time. We cap the maximum distance at 25% of the *dispatch_interval* to prevent bursts of requests to shaped VMs.

```
delta = (error/target) * dispatch_interval;
delta = min(delta, 0.25*dispatch_interval);

If(next_available_dispatch+delta <= current_time + max_delay){
        if(next_available_dispatch<current_time){
                dispatch_request_now();
                 next_available_dispatch = current_time + dispatch_interval;
        }
        else{
                next_available_dispatch = next_available_dispatch + dispatch_interval;
                dispatch request in (next_available_dispatch+delta - current_time);
        }
        update error;
} else reject request;
```

In addition to errors caused by requests' estimated resource consumptions being different from the desired target, a lack of requests to dispatch also causes short term resource underutilization. While it could be thought that once resource allocations are unutilized they are lost forever, in the short-term it is possible for VMs to reclaim them. Other requests already being processed on the VM are getting ahead of their expected finish time due to less workload on the VM. When additional requests are sent to the VM before the deterministic dispatch time, they may slow down already processing requests due to increasing workload. However, as the existing requests were ahead of their expected finish times they can still finish on time.

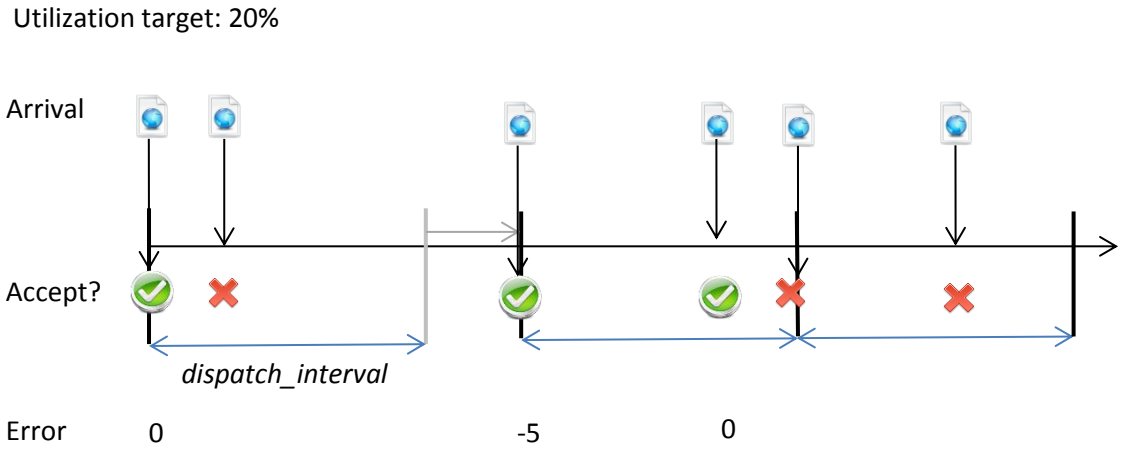


Figure 5-17: Increasing pattern error due to missed request dispatch

```
If(next_available_dispatch < current_time- dispatch_interval)
        error = error + (next_available_dispatch - current_time) / dispatch_interval *
                        target;

delta = (error/target) * dispatch_interval;
delta = min(delta, 0.25*dispatch_interval);

If(next_available_dispatch+delta <= current_time + max_delay){
        if(next_available_dispatch<current_time){
                dispatch_request_now();
                 next_available_dispatch = current_time + dispatch_interval;
        }
        else{
                next_available_dispatch = next_available_dispatch + dispatch_interval;
                dispatch request in (next_available_dispatch+delta - current_time);
        }
        update error;
} else reject request;
```

5.1.4. Page Profiling

To ensure Input Shaper makes intelligent decisions when distributing workload it profiles requests to discover additional information. The information learned is used by the Decision Engine to estimate if a request type is suitable to be included in a shaped pattern. The HTTP shaper module uses the information to estimate if a request will achieve its SLO and if it is correctly achieving VMs' resource utilization targets. Figure 5-18 shows the response time PDF for three of C-MART's pages. It can be clearly seen that the pages take significantly different amounts of time to process. If Input Shaper worked only in a black box fashion its shaping decisions would be much less effective than with its profiled information. For example, My Account pages are much more likely to violate an SLO if they are delayed by the Input Shaper.

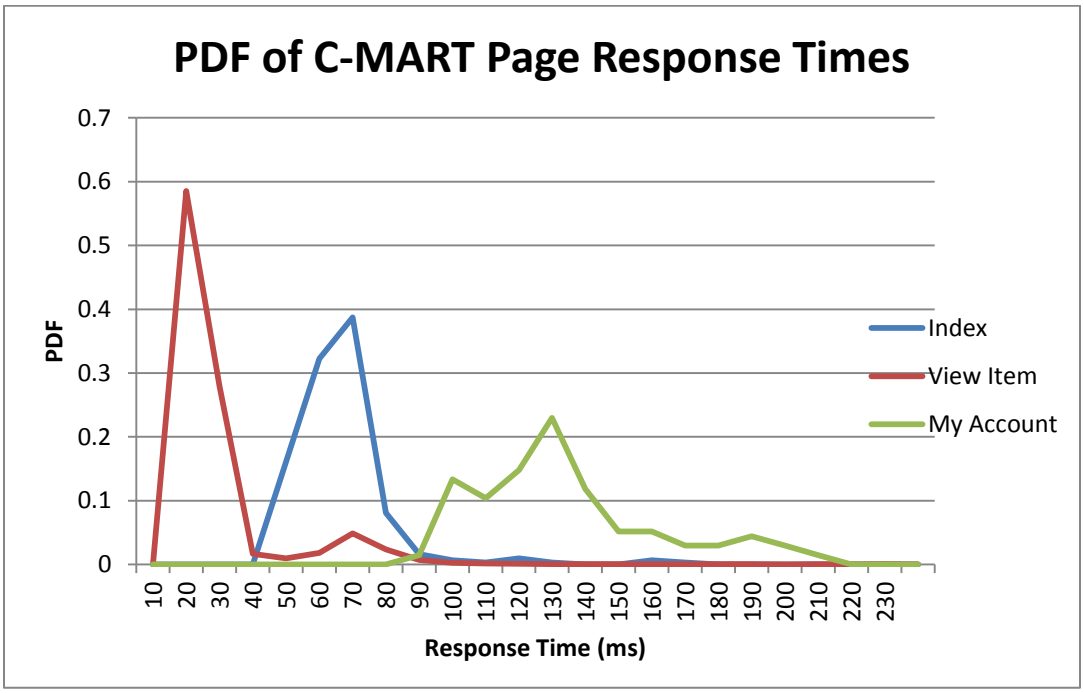


Figure 5-18: PDF of C-MART page response times

To profile requests we create a third datacenter zone isolated from both the shaped and overflow zones. By profiling the requests in isolation we can observe their behavior with minimal interference and achieve the most accurate results. The profiling zone is very similar to the shaped zone. However, the profiling algorithm does not attempt to achieve requests' SLOs. The profiling algorithm has a similar deterministic dispatch pattern to the shaped zone. Rather than attempting to keep a VM at a static resource utilization level, the profiling algorithm periodically alters the number of requests per second that it dispatches. This allows the performance characteristics of requests to be observed at varying resource utilization levels. The profiling algorithm does not have the same error correction for unfulfilled dispatches as the shaped zone. Instead, it buffers requests for up to 2 seconds to ensure that its pattern is always fulfilled. We chose 2 seconds for the default buffering time as it allows a sufficient number of requests to be buffered while not causing catastrophic response time delays to requests that are directed through the profiler. This value can be reconfigured to any period with the use of the Input Shaper's API.

When profiling requests we first use the requests' URLs as their identifiers. We consider each unique URL as a unique request. As some requests are for almost identical data but have different URLs, Input Shaper also offers a grouping function. For example, an image folder may contain 100,000 images with 100,000 different URLs. Rather than considering each image as a separate request type, we instead create group("./images/") that all requests to the images folder will be matched to.

The list of requests available to profile can be accessed via the HTTP shaper module's API. By default, and without any further profiling or configuration, requests are automatically dispatched to the overflow zone. They have their response times, inter-arrival rates, and hit counts monitored. The decision to profile requests is made by the Decision Engine, which then commands the HTTP shaper module to take action.

As we want Input Shaper to begin shaping requests in as short a time as possible, the Decision Engine prioritizes the order in which requests are profiled. For example, we gain more benefit from profiling a request that has one thousand arrivals per minute compared to a request that has ten arrivals. The priority becomes less clear when requests have similar hit rates, for example a request that arrived 450 times versus a request that arrived 500 times. The Decision Engine ranks requests based upon the following criteria:

Hit count: We want to profile requests that we see a large number of as we will shape the majority of the traffic in a shorter time

Large difference in %-tile request RT and SLO RT: The longer we can delay a request the easier we can fit it to a shaped pattern

Low response time std. dev.: We want to profile requests that have predictable response times as it can be related to resource utilization. A predictable response time can indicate predictable resource utilization

$$\alpha \frac{request_{hits}^{i}}{\sum_{j \neq i}^{n} request_{hits}^{j}} + \beta SLO_{RT} - request_{RT_{CDF}}^{i}{}^{-1}(SLO_{\%}) + \gamma std.dev(request_{RT_{CDF}}^{i})$$

We currently normalize all three values and choose the request with the highest score as the request to be profiled. During request profiling the Decision Engine collects the isolated VMs' resource utilization levels from the Dynamic Resource Controller. This allows the average resource consumption per request to be calculated. We later use this value when creating VMs' request patterns to calculate the number of requests per second required to satisfy a VM's target resource utilization.

In addition to profiling requests based on their URL, the profiler can also split requests based on the HTTP protocol type or parameters in their URLs. This is extremely useful for pages such as C-MART's Image Upload page. A client performing a GET on the page receives a simple HTML page as a response, which takes only a few milliseconds to complete. A client performing a POST on the page may be uploading multiple Megabytes of data, which can take multiple seconds to complete. The Decision Engine can detect such behaviors by looking for multiple response time peaks in requests' response time distributions. It can then set the profiling level that it wishes to perform; e.g. only /cmart/upload.html-GET. If it finds that a request should be differentiated on data other than its URL it is set in the request's properties. Future incoming requests are then split into two request types based on the differentiation criteria.

5.2. Implementation

In this section we provide an overview of the Input Shaper's implementation. It is implemented as a module of Apache HTTP Server. This allows it to work with generic web applications on a large scale; it is not just a toy example implementation. We choose this approach as it provides us with results that are representative of production environments which is preferable to results that are only attainable in a research lab. It also allows Input Shaper to integrate with other modules and features that are offered by the Apache HTTP Server application. For example, we utilize the module 'mod_form' that allows inspection of HTTP requests' form data [82]. This data is leveraged by the request profiler.

The implementation of the HTTP Shaper module is written in C and is 4,000 lines of code. We have to implement in C as this is the language that Apache Server's modules must use. Using C allows the Input Shaper to achieve high performance; each request takes less than 0.1ms to process in the Input Shaper module. A fast processing time is imperative as every incoming request is affected by the additional processing delay of the Input Shaper. During our experimentation the additional processing time of the Input Shaper module is negligible. The Input Shaper's Decision Engine is another 8,000 lines of Java code than can be executed on a different host.

To integrate Input Shaper with an Apache HTTP Server our mod_proxy_inputshaper.so is added to the server's module directory. In addition, a modified version of mod_proxy_balancer.so is required to integrate the Input Shaper's dispatch algorithms with the

regular load balancer algorithms. The only modification of the regular mod_proxy_balancer.c is adding the Input Shaper balancer method:

```
ap_register_provider(p, PROXY_LBMETHOD, "byinputshaper", "0", &byinputshaper)
```

To start using an application with Input Shaper we only need to change the application's load balancing method to 'byinputshaper' in the httpd.conf file:

```
<Proxy balancer://mycluster>
        BalancerMember http://192.168.1.50:80
        BalancerMember http://192.168.1.51:80
        ProxySet lbmethod=inputshaper
</Proxy>
ProxyPass /test balancer://mycluster
```

By default, the Input Shaper will not shape incoming requests using its deterministic algorithm. It will instead dispatch all requests via a round robin algorithm to all of the BalancerMembers. The requests and balancer members to shape are configured via the Input Shaper's API discussed later in section 5.2.3.

Figure 5-19 shows an overview of data structures used in the Input Shaper module. Each request runs in an isolated Thread. This is an Apache HTTP Server design pattern that provides isolation between processing requests. If one request causes a segfault or other runtime error it is the only request affected by the error.

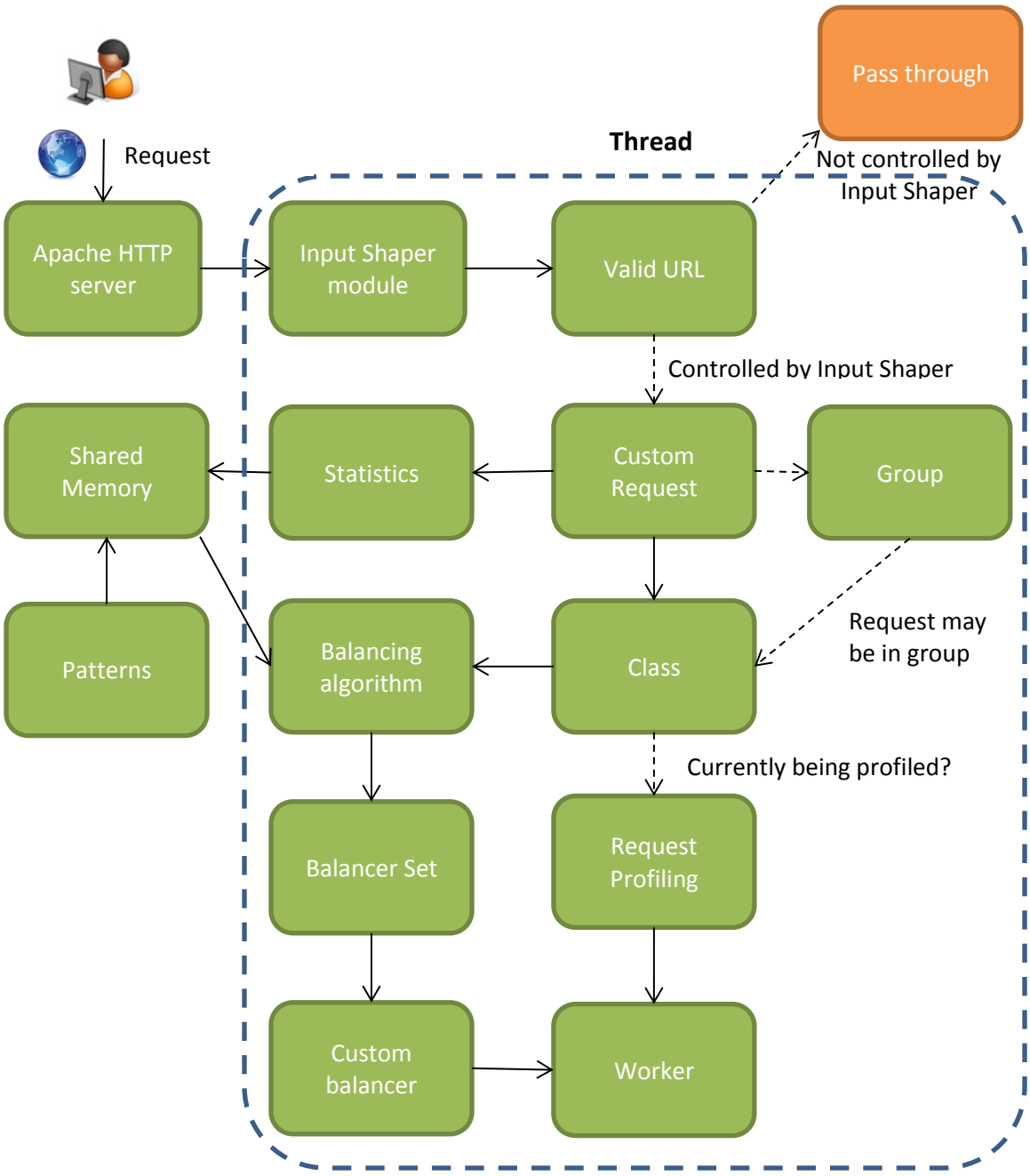


Figure 5-19: Data structures used in the Input Shaper module

5.2.1. Request Shaping

If a request's URL matches a pattern that the Input Shaper module has been told to shape it is admitted in to the Input Shaper module. The request is then further processed to identify if it is assigned to any group or class. Each request is first identified by its URL. The properties for that URL pattern are retrieved from memory. If there is no result returned then the request is of a type we have not previously observed or grouped, and a new properties entry is created. If properties are returned they are checked to see if any additional filtering should be performed on the request. For example, we may split a request to "/upload.html" into GET and POST classes. The request is repeatedly filtered until the properties returned are the most specific match for the request. A request's properties provide information on the id, resource consumption, and class of a request.

After retrieving a request's properties the Input Shaper module decides how the request should be dispatched. The request is compared to each configured dispatch pattern to check if the pattern would like to consume the request. If a pattern would like to dispatch a request it sets the request's 'worker candidate' and dispatch_time variables. The worker candidate is the IP address that the request is dispatched to. Once the worker candidate is set we know that a request has matched to a pattern and we do not need to compare it to the remainder of the patterns. If a pattern does not wish to admit a request it simply leaves the worker candidate variable null and returns.

For a pattern to decide if it will admit a request or not it needs to compare the request to its configured dispatch pattern. To allow reconfiguration of patterns at runtime the balancing

algorithm and pattern properties are all variables. A case statement is used to run the correct algorithm. The default case of 'find_best_byrequests_custom' is the Input Shaper's modified version of the default round robin algorithm

```
static proxy_worker *run_selected_algorithm(proxy_balancer *balancer, request_rec *r,
is_scfg *scfg, is_rcfg *rcfg, is_pattern* pattern){


        switch (pattern->lb_algorithm) {
                case -1: {
                        return find_best_example(balancer, r, scfg, pattern);
                        break;
                }
                case 0: {
                        return find_best_byrequests_custom(balancer, r, scfg, pattern);
                        break;
                }
                case 1: {
                        return find_best_learn(balancer, r, scfg, pattern);
                        break;
                }
                case 2: {
                        return find_best_det(balancer, r, scfg, pattern);
                        break;
                }

                default:{
                        ap_log_error(APLOG_MARK, APLOG_CRIT, 0, NULL,
                                "IS run_selected_algorithm: no alg. for %s and url %s",
                                pattern ->description, r->uri);
                        return NULL;
                }
        }
}
```

The pattern matching decision for each request must be performed sequentially. If multiple requests are being compared to the same pattern in parallel multiple may match and be assigned the same dispatch slot. To prevent this the proxy balancer module runs as a single thread. Each Apache child thread that is processing a request waits in turn to compare to each pattern, as shown in Figure 5-20.

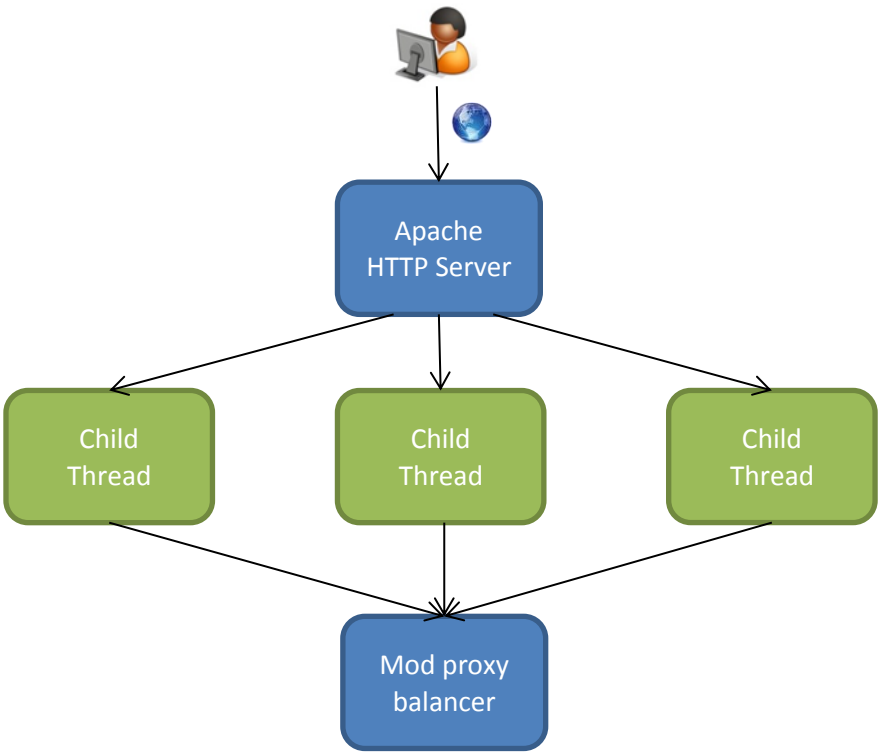


Figure 5-20: Input Shaper's parallel and sequential processing

5.2.2. Shared Memory Problem

Apache Server's child processing threads are isolated from each other to prevent errors in one request's processing affecting other requests. This is beneficial from a security and stability point of view. However, it causes difficulties for the Input Shaper as patterns, properties, and statistics need to be accessible from all child threads. To mittigate this problem we use the Apache tools shared memory routines. A shared memory object is created by allocating memory, and then making its address available to other threads. For each shared memory location a file is created on disk with the address of the shared memory. A file on disk is used as there is no fixed memory locations know by each child thread to lookup the shared memory's location. However, they can all access a fixed file location on disk. Each child thread can then create a pointer to the shared memory objects, as illustrated in Figure 5-21.

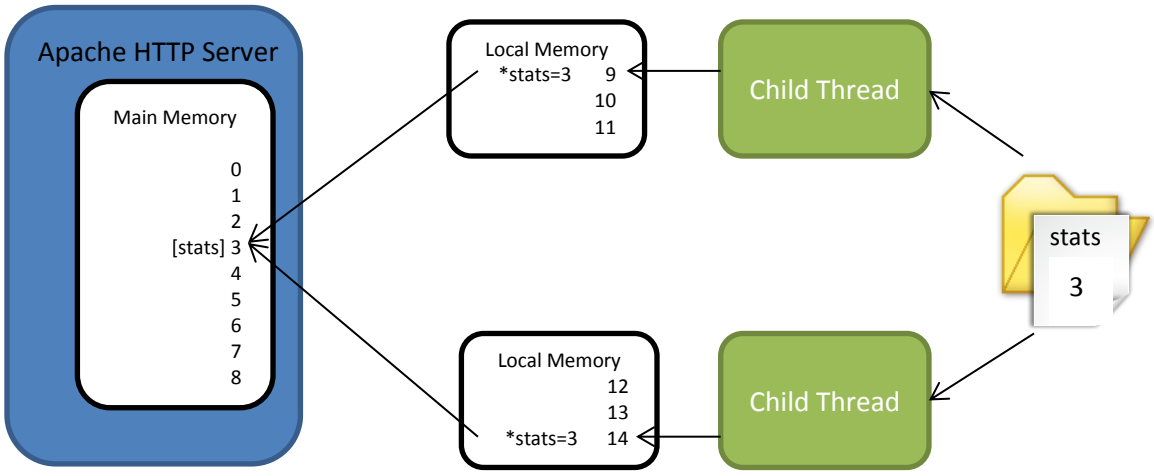


Figure 5-21: Sharing memory between Input Shaper threads

Below we show the process of allocating and reading a shared memory address:

```
if (!scfg->common_shm) {
        rv = apr_shm_attach(&scfg->common_shm, scfg->isCommonSHMFile, p);
        if (rv != APR_SUCCESS) {
                ap_log_error(APLOG_MARK, APLOG_CRIT, rv, s, "Failed to allocate ");
                return;
                }
        }

        // Get the local address for the shared segment
        scfg->stats = apr_shm_baseaddr_get(scfg->stats_shm);
        scfg->common = apr_shm_baseaddr_get(scfg->common_shm);
}
```

5.2.3. Input Shaper API

To separate data processing and analysis from the following of patterns we separate the Input Shaper's HTTP module from the Input Shaper's Decision Engine. To allow the Decision Engine to control the HTTP module we implemented an API that exposes all of the HTTP module's functionality. As the HTTP module is running as part of a web server, the most logical solution is to implement the API over HTTP – which it is. The HTTP module filters requests' URLs for the base server URL of "/inputshaper". A request sent to this URL is processed by the HTTP module as a command. If the command is not valid an error code is returned. If the request is valid, results from the execution of the command are returned. If there are no results "OK" is returned if command completes successfully. Table 5-3 provides an overview of Input Shaper's HTTP model's most important API commands.

Command	Parameters	Description
addMonitorURL	**url** – the url to monitor	Adds a URL that the Input Shaper should begin monitoring. All child URLs are also monitored. I.e. /app1/ also monitors /app1/folder2/page.html
addNonMonitoredURL	**url** – the url to ignore	Adds a URL that the Input Shaper should ignore. All child URLs are also ignored. I.e. /images/ also ignores /images/home/icon.png
getMonitorURL	*none*	Returns a list of URLs that are being monitored
getNonMonitoredURL	*none*	Returns a list of URLs that are not being ignored
getWorkers	*none*	Returns a list of available HTTP workers. A worker is a VM that can process a request for a given application
getBalancers	*none*	Returns a list of Apache Balancer objects. An Apache balancer is the entrance point for clients to access applications. Each application typically has one balancer
getCustomBalancers	*none*	Returns a list of custom balancers. A custom balancer is Input Shaper's own balancer object that allows the splitting of applications into multiple zones – shaped, overflow, profiling
createCustomBalancer	**name** – identifies of custom balancer **index** – Its place in the custom balancer list	Creates our custom balancer object that will contain multiple worker VMs
editCustomBalancer	**index** – element to edit **worker** – The VM IP to add to balancer **balancer** – The app the worker belongs to	Edits a custom balancer to add a new worker VM. The VMs will have requests dispatched to them depending on their patterns

Command	Parameters	Description
setPattern	**id**: The pattern's index **classid**: Which class to admit **workerid**: The worker to send to if matched **rps**: Desired requests per second **target**: The target utilization	Adds a pattern to a worker VM. Requests are compared to the pattern and dispatched to the worker VM if they match
getArrivalDistributions	none	
setProfile	**page_index**: The index of a page to profile **class_index**: The index of a class to profile **custom_bal_id**: The index of the customer balancer to dispatch traffic to	Sets that a page or class should begin profiling. The custom balancer is the 'zone' that is used for profiling
pauseProfile	*none*	Stop redirecting requests to the profiler, but do not delete any of the profiler conf. or data
stopProfile	*none*	Stop profiling requests and clear the configuration
setProfileLevel	**level**: How requests should be spit	How requests should be categorized. 0=by URL, 1=by URL+HTTP type, 2=by form data, 3=all of above
getProfilingStatistics	*none*	Returns the response time and arrival distributions for the profiling object
addURLToGroup	**url**: The URL to add **group_id**: The group to add URL to	Adds a URL to a group so, for example, requests for the same folder will be grouped
configureRequest	**util**: Avg. consumed	Set avg. resources consumed
getClasses	*none*	Returns a list of classes
addToClass	**id**: request to add to class **class_id**: that class to add to	Add a request to a class
clearStatistics	*none*	Clears stale statistics

Table 5-3: List of Input Shaper's API commands

5.3. Results

5.3.1. Experimental Setup

Our experiments are performed on a flat local area network using commodity hardware. This is similar to the hardware found in cloud computing environments. We use KVM as our hypervisor. The datacenter cloud hosts for the Input Shaping zones consist of six Intel i7-3370 3.4 GHz quad-core servers with 8GB of RAM. The other hosts are four AMD FX4100 3.6GHz quad-core servers with 8GB of RAM. Each VM has its virtual hard disk on the hosts' local hard disk. We use a flat network topology with each host connected to the switch with a 1GB/s network interface.

C-MART is configured with three application VMs and four MySQL VMs. There is one application and data tier per shaping area, and an additional MySQL master for write replication. This is a typical configuration for scaling MySQL with master/slave replication. Each VM is placed on its own host to prevent the application causing performance degradations to itself. It is expected that applications' VMs would not share the same host as their workload levels are correlated. If we place multiple VMs for the same application on a single host we would lose the benefits of statistical multiplexing workloads.

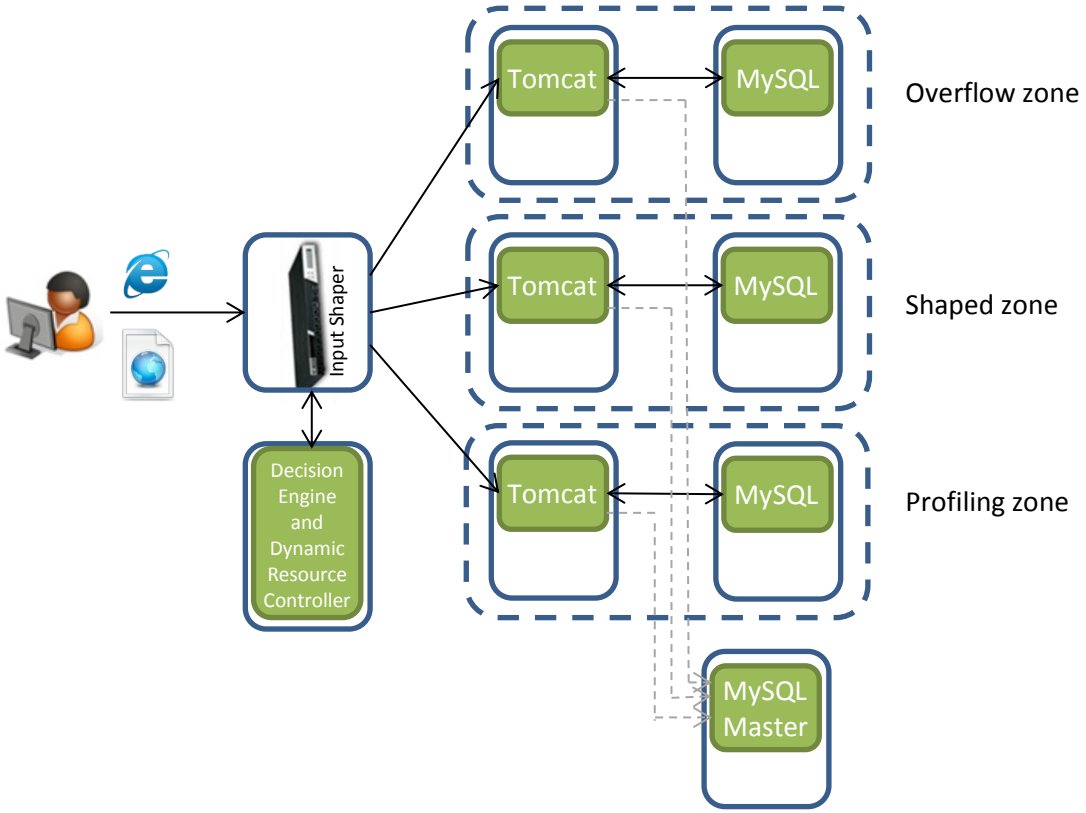


Figure 5-22: Experimental setup for Input Shaper

5.3.2. C-MART Shaping

To test the effectiveness of Input Shaper we run C-MART with and without Input Shaper enabled to compare the results. With Input Shaper disabled, clients' requests pass straight through to the application tier in the 'shaped zone'. This provides us with a baseline reading of C-MART's traffic. With Input Shaper enabled, clients' requests are shaped and sent to either the VM in the shaped zone, or a second VM in the overflow zone. We monitor the CPU utilization of the host in the shaped zone where the C-MART application VM is located. As we use the same number of clients for each experiment, the total amount of resources consumed is equal.

However, the exact distribution of when resources are consumed will differ depending on clients' behaviors. During the experiment we are concerned with whether Input Shaper allows us to control and reduce variance in VMs' resource utilization levels. This would allow a reduction in applications' own, and other co-located VMs', resource overprovisioning.

5.3.3. CPU Utilization

Figure 5-23 shows the shaped Tomcat host's CPU utilization with Input Shaper enabled and disabled. It clearly shows that with Input Shaper enabled the host's CPU utilization has less variance. With Input Shaper disabled the variance is 147. This is eight times greater than the variance of 17 when it is enabled. This is the result we would expect for the shaped host as Input Shaper is redirecting excess load away from the shaped host to the overflow zone.

Table 5-4 shows the host's average CPU utilization with Input Shaper enabled and disabled. The average CPU utilization with Input Shaper enabled is 6% lower than when it is disabled. The difference is due to the load that is instead being sent to the overflow zone. This is expected as it is unlikely that all incoming requests will conform to the desired request pattern. During our experiment the shaped zone processed 81% of the incoming workload.

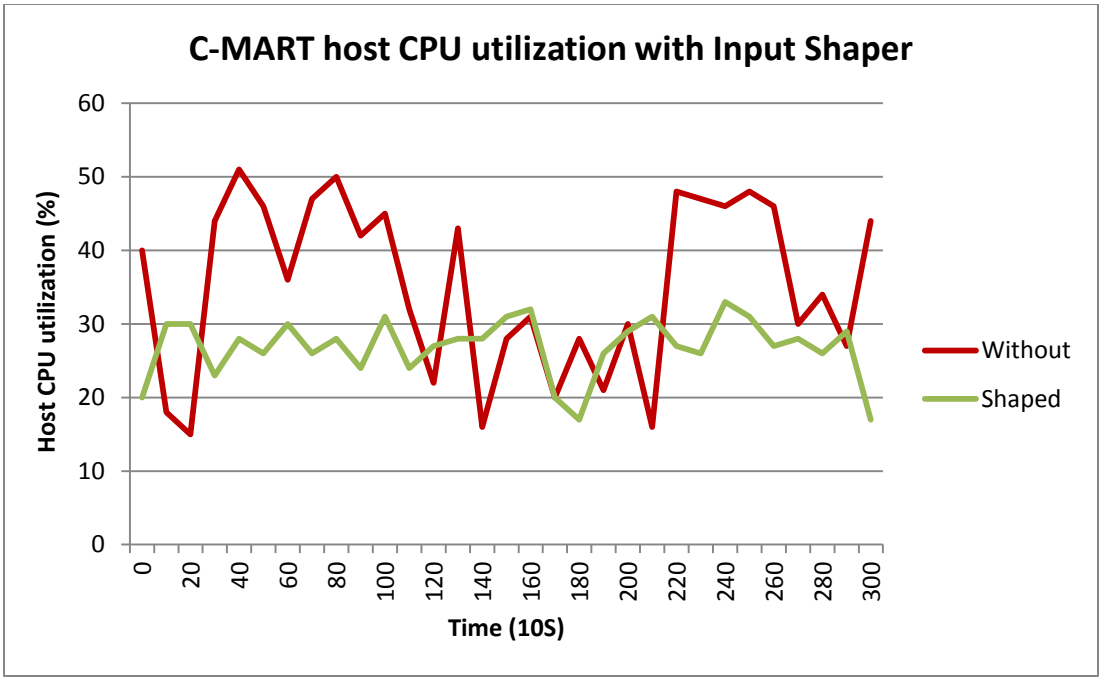


Figure 5-23: CPU utilization with Input Shaper enabled or disabled

Input Shaper	Avg. CPU Util	Max Util	Variance	Std. Dev.	Max – Avg. Util
Disabled	33%	51%	147	12%	17%
Enabled	27%	33%	17	4%	6%

Table 5-4: CPU utilization metrics when using Input Shaper

The shaped host's resource utilization is not completely static due to the arrival rate and workload mix of the incoming traffic. If there is not sufficient workload arriving for the application then the resource utilization will always decrease. For example, during the period between 10 and 30 seconds for the unshaped host the resource utilization drops to 15%. Even if we were shaping the traffic during that time additional load would not be available to increase the resource utilization. Similarly, if incoming requests arrive in close proximity to each other some will be sent to the overflow zone. Even if the aggregate resource requirements for

incoming requests are sufficient to fully utilize the shaped host, their arrival rate may not be suitable to allow them all to be dispatched to the shaped host. This is because the Input Shaper has a rate limiter to prevent shaped hosts receiving large bursts of requests and a maximum request wait time.

5.3.4. Reduced Resource Waste

A benefit of shaping applications' workloads is reducing resource waste. The first way that waste is reduced is by reducing utilization variance to keep VMs as close to their desired resource allocations as possible. A simple resource allocation scheme is to allocate VMs' resources based upon their maximum resource utilization level. This ensures that they always have sufficient CPU cycles available to fulfill their utilization requirement. Using the data in Table 5-4 would result in a 51% and 33% CPU allocation for the unshaped and shaped workload respectively.

Figure 5-24 illustrates the amount of time that the VM is utilizing less than its resource allocation based on the maximum allocation scheme. It can clearly be seen that when the VM is running with Input Shaper disabled it is on average much further from its maximum resource utilization level. With Input Shaper disabled there are 549 Billion CPU cycles unutilized compared to its allocation. With Input Shaper enabled only 137 Billion CPU cycles are unutilized compared to its allocation. This results in a 75% reduction in the amount of allocated resources that are unused in the shaped zone. This result does **not** indicate that 75% fewer resources are required to run the application, as not all of its allocation is waste. It does however show that resource overprovisioning can be reduced, thereby increasing hosts' average resource

utilization. Section 5.3.5 discusses how this reduction translates into reduced resource requirements for cloud environments.

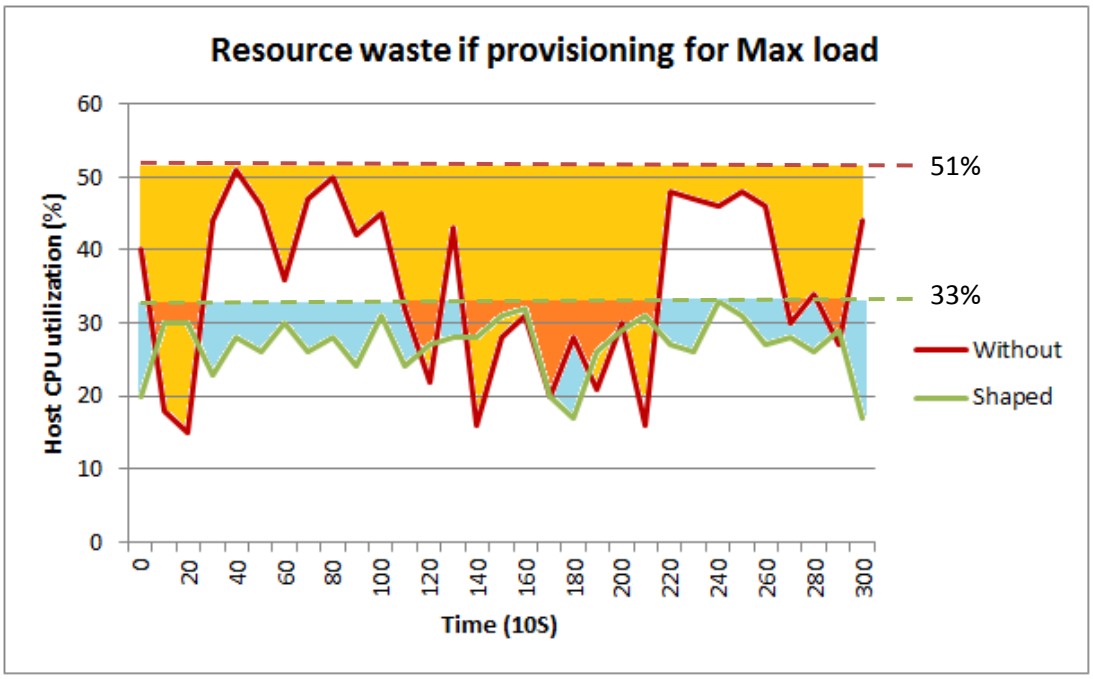


Figure 5-24: Amount of resource waste with and without Input Shaper

5.3.5. Total Resource Requirements

In addition to shaping workloads to reduce resource utilization variance and resource waste, significant resource utilization improvements can be achieved by reducing the amount of resources that remain unallocated. For example, if a VM is only using 50% of a host's available resource allocation, it is desirable to place another VM on the host to consume the remaining resources; as previously shown in Figure 5-10.

To calculate the potential resource savings that Input Shaping can achieve we consider that we have multiple applications running that are similar to those described in Table 5-4. We

then place the applications' Tomcat VMs on hosts and compare the total amount of resources required to satisfactorily run the applications. Figure 5-25 shows the resources required for nine instances of the application running with and without Input Shaping. The placement scheme we use is the maximum resource utilization scheme where VMs are ensured to receive their maximum CPU demand.

Figure 5-25 represents the worst-case scenario for VM workloads when no Input Shaping is available. As the VMs' maximum CPU utilization level is 51% each host requires its own host under the maximum resource allocation scheme. This causes 49% of the datacenter's resources to be unallocated and therefore wasted. Input Shaping allows the VM's resource demands to fit much better with the resources available. Rather than 49% of resources being unallocated on nine hosts, only 1% remains unallocated on each of the three shaped hosts, and a total of 47% on the two overflow hosts. This is a reduction from nine hosts to five hosts to satisfy all of the VM's resource demands; a 45% reduction in the number of hosts required.

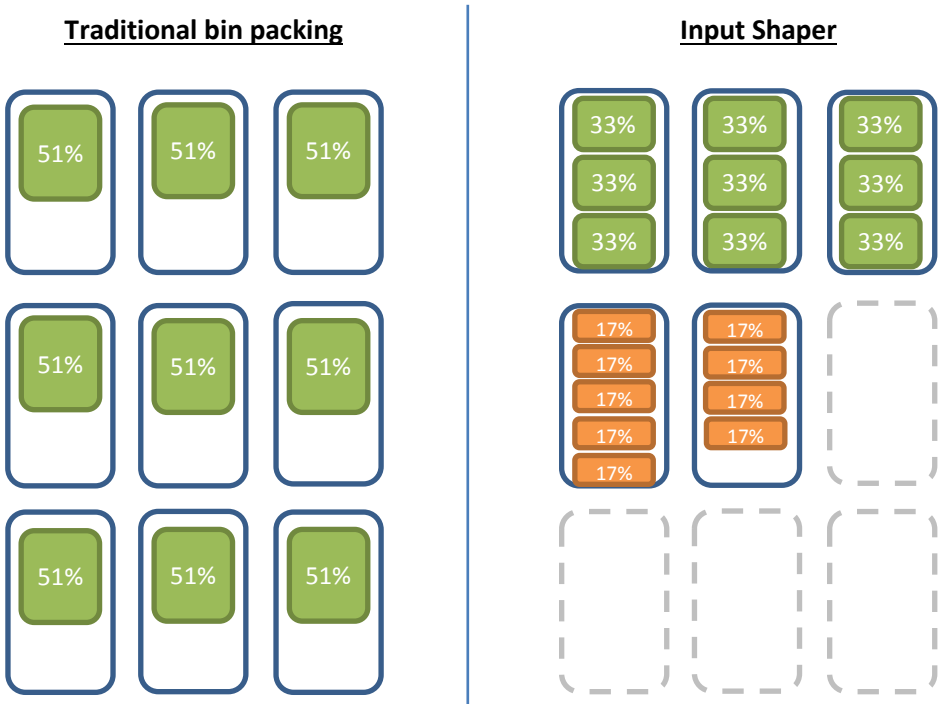


Figure 5-25: Resources required for shaped and unshaped VMs

The resource allocation levels shown in Figure 5-25 are the maximum resource utilization values for each VM. Using the data in Table 5-4 we can see that the average resource utilization levels are 33% and 26% with and without Input Shaper respectively. Using these values we can calculate the average hosts' resource utilization level for each scheme in Figure 5-25. Using Input Shaper allows an average resource utilization level of 57%, versus 33% without; a 72% improvement.

$$Average\ Resource\ Utilization = \frac{\sum_{i=0}^{n} host^{i}_{resource\ utilization}}{\sum_{i=0}^{n} host^{i}_{total\ resources}}$$

$$Traditional = \frac{33\% * 9\ hosts}{9\ hosts}$$

$$= \frac{0.33 * 900}{900}$$

$$= 33\%$$

$$Input\ Shaper = \frac{(26\% * 3) * 3\ hosts + (6\% * 5) * 1\ host + (6\% * 4) * 1\ host}{5\ hosts}$$

$$= \frac{234 + 30 + 24}{500}$$

$$= 57\%$$

Although the VM placement in Figure 5-25 shows a 72% improvement in resource utilization when using the Input Shaper, it does not consider the expected statistical multiplexing of the overflow zone. As the overflow zone is not designed to always achieve applications' SLOs, it would not allocate VMs' resources based upon their maximum requirement. Instead the Dynamic Resource Controller would alter VMs' resource allocations as workloads vary. Figure 5-26 shows the VM placement of the datacenter if we instead place the VMs in the overflow zone based upon their average resource utilizations. In this scenario we only need four servers to host all of the VMs. This gives us a final average resource utilization level of 72%, a 118% increase over the maximum allocation scheme.

$$Input\ Shaper = \frac{(26\% * 3) * 3\ hosts + (6\% * 9) * 1\ host}{4\ hosts}$$

$$= \frac{234 + 54}{400}$$

$$= 72\%$$

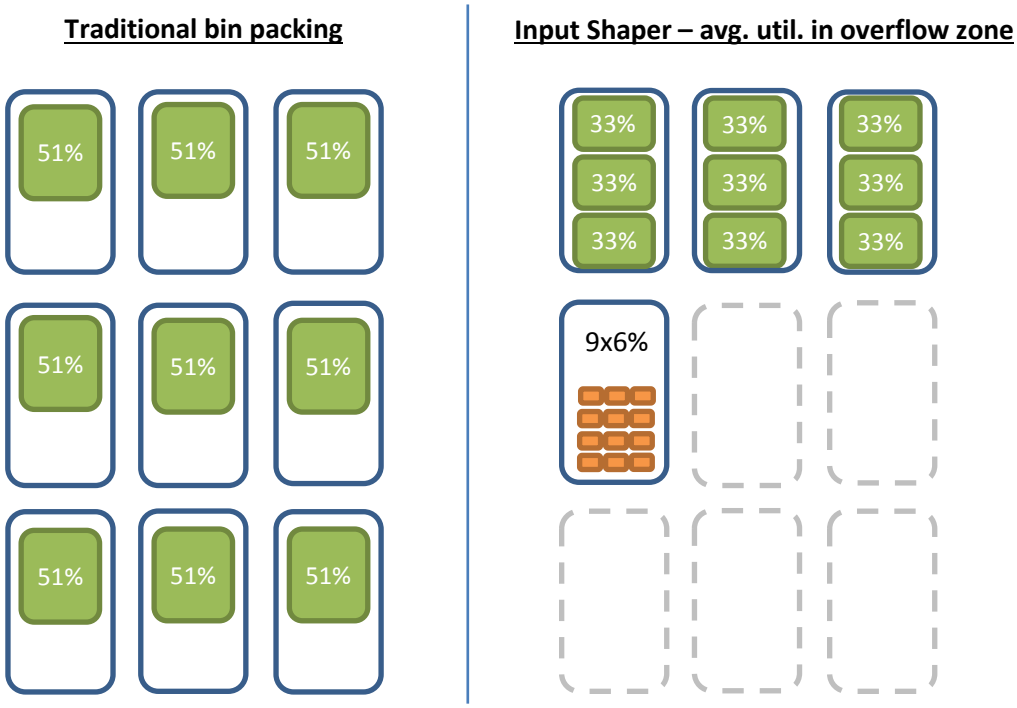


Figure 5-26: Statistical multiplexing of overflow zone

For the previous calculations on Input Shaper's improvement in datacenter's resource utilization levels we have considered the datacenter's worst-case scenario, where VMs' resource requirements are only slightly too high to consolidate them onto a single host. Figure 5-27 shows that when the maximum resource utilization of the VMs is only 1% lower, a traditional bin packing scheme is able to consolidate them much more efficiently than before. In Figure

5-27 there are 16 replicas of the service running. This is the worst-case scenario for Input Shaper as it removes the significant gains achieved by reshaping VMs' resource requirements to fit those available on the hosts. At a 50% resource requirement per VM they already consolidate perfectly onto available hosts.

Although in this case gains cannot be achieved by directly better fitting the VMs' resource requirements to the hosts, the reduction in overprovisioning and the statistical multiplexing of the overflow zone still allow some reduction in resource requirements. The traditional bin packing uses 8 hosts, while the Input Shaper uses 7 hosts. Also, due to the reduction in resource overprovisioning, the 7th host still has 67% of its resources free to allocate if another VM is added to the shaped zone. Using the 33% and 26% average VM resource utilizations from Table 5-4, the average resource utilization for each scheme is 66% and 73%; a 10% improvement when using Input Shaper.

$$Traditional = \frac{(33\% * 2) * 8\ hosts}{800} = \frac{528}{800} = 66\%$$

$$Input\ Shaper = \frac{(26\% * 3) * 5 + (27\% * 1) * 1 + (6\% * 16) * 1}{700} = \frac{513}{700} = 73\%$$

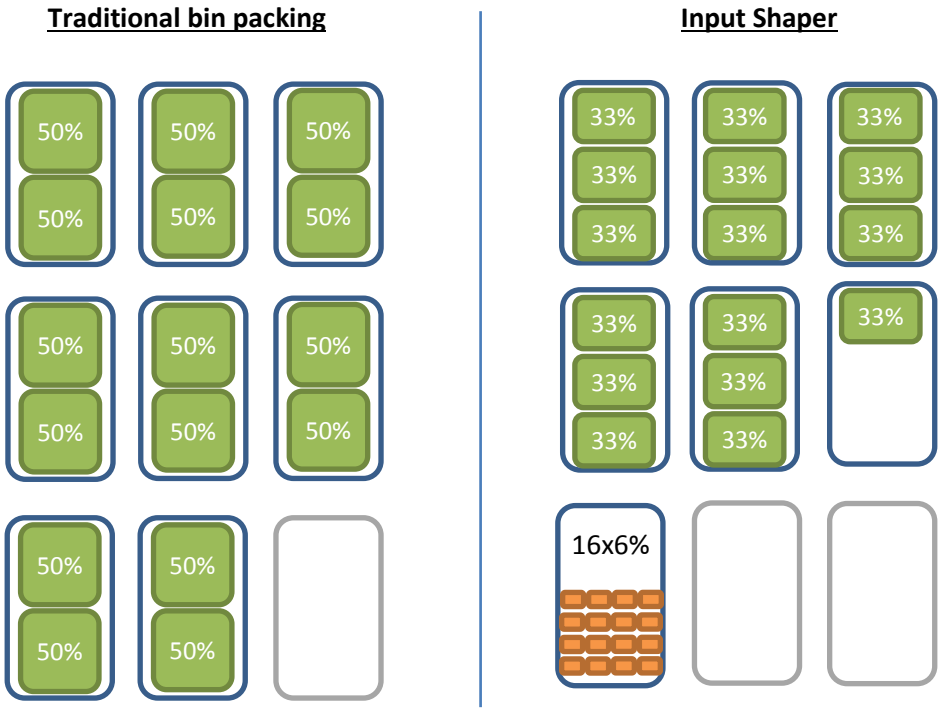


Figure 5-27: Shaping already densely packed VMs

5.4. Conclusion

In this chapter we show that shaping applications' incoming workloads can improve the resource utilization of cloud computing environments. By reducing the variance of VMs' resource utilization levels we can reduce the amount of resource overprovisioning required for them to achieve their SLOs. Also, it reduces the risk that co-located VMs fail to achieve their SLOs by reducing the variance in resource contention. In addition, Input Shaper can shape the resource requirements of VMs to better fit the existing resources available in the datacenter. This allows hosts to be packed more densely with VMs, reducing the amount of unallocated resources, and reducing the number of servers required to host the VMs.

We evaluate Input Shaper by runner the C-MART benchmark with Input Shaper both enabled and disabled. The results confirm that the variations in hosts' resource utilizations in the shaped zone are reduced from 147 to 17, an 88% reduction. This reduces the amount of resources wasted due to resource overprovisioning by 75%. By shaping VMs' resource requirements we show that the number of servers required to host applications can be reduced by 45%.

6. Conclusion

In this thesis we propose a joint dynamic resource allocation and workload shaping cloud management system. By integrating these two systems we can achieve applications' SLOs where current schemes would fail. Without control over VMs' incoming workload levels, current schemes must significantly overprovision applications' resource allocations, and even then cannot guarantee that SLOs are achieved. Without dynamic resource allocation an admission control or load balancing system cannot increase applications' resource allocations when their workload levels increase. This will cause a high number of requests to be dropped or cause SLO violations. We present Dynamic Resource Controller that allocates multitier applications the correct amount of resources they require to achieve their SLOs. We also present Input Shaper that allows applications' incoming workloads to be shaped such that hosts' resources are fully utilized, thus improving the datacenter's efficiency. In addition, Input Shaper reduces the variance in VMs' workload levels. This prevents hosts from becoming overloaded and causing SLO violations, while also reducing the amount of resource overprovisioning.

6.1. C-MART

To evaluate cloud management systems we present C-MART, an application benchmark designed to mimic the behavior of production applications running in cloud environments. The more varied and dynamic characteristics of C-MART compared to previous benchmarks ensure that cloud management schemes are rigorously evaluated. C-MART can expose previously unidentified problems or inaccuracies in currently proposed management schemes.

We show that C-MART can identify a 22% failure rate in SLO achievement where a previous benchmarks shows a 0% failure rate. The increased failure rate is due to the more varied resource utilization of C-MART's servers. Due to C-MART's page complexity we also show a 1040% increase in error when predicting VMs' resource utilization levels when using a simple request arrival rate prediction scheme.

We believe that the design and implementation of C-MART are a good example of how future benchmark applications should be created. Without the complex behaviors and interactions of realistic applications benchmarks produce overoptimistic results. This results in systems under test being poorly evaluated and prevents benefits identified in the research environment being achieved in production environments. In addition, by only designing schemes that perform well using simplistic benchmarks, researchers may limit their own contributions by failing to identify interesting data that can be exploited to their benefit.

By designing C-MART as a realistic cloud application we have shown the complexity involved in creating a realistic benchmark. C-MART uses many different programs and technologies to provide the user with a seemingly simplistic online auction system. This shows the importance that usability and flexibility have in benchmark systems. As different business will make different design decisions when creating their applications a benchmark should be able to emulate this. Benchmarks should also be able to evaluate systems under test using different configurations and scenarios to ensure the system performs satisfactorily under many circumstances.

6.2. Dynamic Resource Controller

Unlike previous dynamic resource allocation schemes our system does not attempt to normalize or evenly distribute performance between applications. It instead achieves application-level response time based SLOs. This is a more desirable goal as end users' perceptions are based on the application-level service they observe, and not the equality that other users receive or the resource utilization of a remote server. Our scheme also considers the effect of resource contention on applications' performances; which can be significant.

We show that our Dynamic Resource Controller can improve applications' performances by 30% compared to static resource allocation schemes. In addition, we show that applications' resource requirements can be decreased by 20%. These gains are achieved by dynamically allocating hosts' resources where they are most needed depending on applications' workload levels and resource requirements.

We conclude that allocating VMs static resource levels is insufficient and wasteful when attempting to achieve application-level SLOs. Resources must be allocated in coordination across all application tiers to ensure satisfactory end-to-end response time. If the differing resource requirements of each tier are not considered a high amount of overprovisioning occurs. As there are many combinations of resource allocations and utilizations across different tiers a mechanism to reduce the dimensionality of the problem is also required. In addition, as conditions in the datacenter can quickly vary having historical system state to make calculations can drastically reduce the time taken to converge to a suitable resource allocation.

6.3. Input Shaper

As opposed to current cloud management schemes that only react to applications' workload levels we present Input Shaper which actively shapes applications' incoming workloads. It reduces variations in VMs' workload and resource utilization levels to near deterministic levels. This results in a reduction in resource overprovisioning of 75%. Input Shaper also provides control over VMs' resource requirements by controlling their workload levels. This allows VMs' resource requirements to match the resources available on hosts ensuring that VMs are densely packed, reducing the number of hosts required by up to 45%.

We conclude that shaping applications' input provides many benefits for controlling application-level performance and is required to reliably achieve SLO. Without a method to control applications' inputs there is no mechanism to prevent applications becoming overloaded. Due to variations in requests' resource consumptions and processing times load balancing via black-box methods results in significant resource overprovisioning and does not guarantee request response times. As the variation in resource requirements can vary by multiple orders of magnitude, profiling requests vastly improves the accuracy with which VMs' resource utilization levels can be estimated.

We also believe that segregating the datacenter workloads in to multiple isolated areas provides significant benefits. It ensures that problem and high variance workloads do not interfere with other applications' performances. By segregating low and high variance workloads we ensure that both VMs resource utilization and resource contention levels are constrained. This decreases the probability of SLO violations.

6.4. Future Work

In this section we present future work that could benefit cloud management systems. The management of cloud computing environments has such great scope and importance that further research is warranted.

6.4.1. Further Investigation Into Request Dispatch Patterns

In this work we focused on deterministic dispatch patterns. As we are attempt to reduce variances in VMs' workload and resource utilization levels this is a logical approach. However, as requests' datacenter arrival times and resource consumptions are not deterministic, other request dispatch patterns may produce lower overall resource utilization variance. For example, it may be possible to use requests' response times to estimate their resource consumptions. If one request of type A takes 100ms to process and another request of type A takes 120ms to process we may conclude that the second request consumed more resources to process. This is a common behavior for applications' requests as the exact data processed for the same request type can differ. When accessing C-MART's My Account page the requests' resource consumptions vary based on the number of previous items the user has bid on. It may therefore be possible to use requests' response times to estimate when the next requests should be dispatched to a host.

Another way that requests' response times could be used is to identify temporary VM overload. If we dispatch multiple larger than average requests in multiple consecutive dispatch periods we may temporarily cause an increase in requests' response times. We could use

response time feedback to identify the overload and temporally reduce the number of requests dispatched to reduce the duration of the overload.

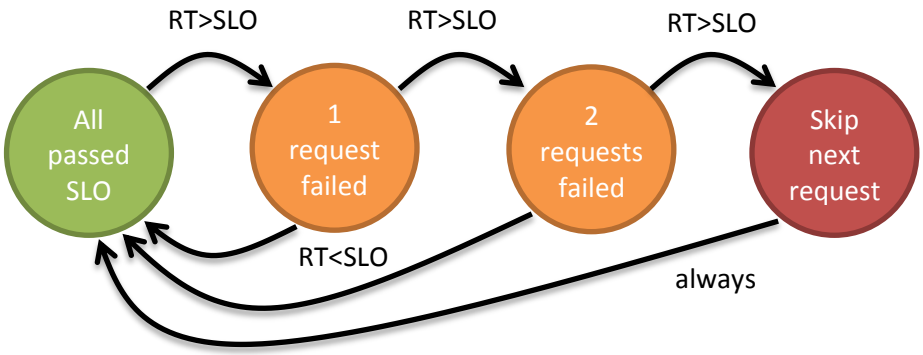


Figure 6-1: Using response time feedback to reduce overload

6.4.2. Shaping Multiple Technologies Using a Single Input Shaper

We have implemented an Input Shaper for HTTP requests within applications. However, applications utilize many different protocols and technologies. Each of these may have its own load balancing system. To reduce the workload and resource utilization variance in a greater number of VMs we could integrate the load balancing of different systems into a single Input Shaper system.

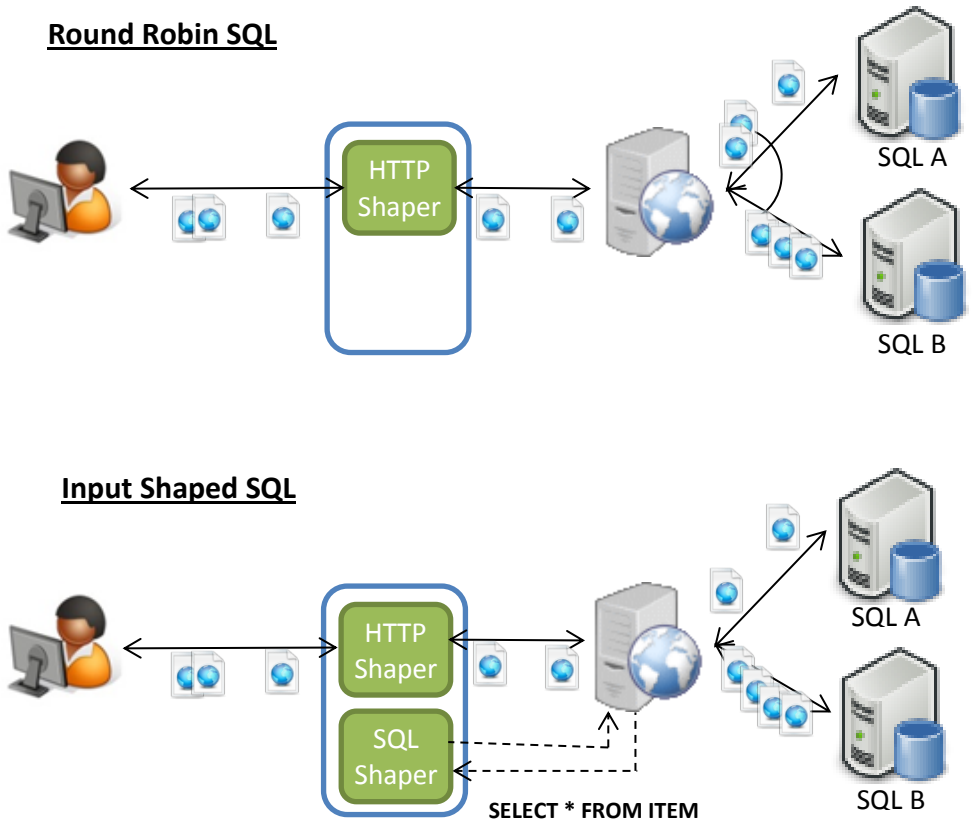


Figure 6-2: Input Shaping multiple technologies

Figure 6-2 shows how Input Shaping could work with both HTTP and SQL Input Shapers. Rather than shaping traffic at the HTTP tier only, the application could shape traffic at the HTTP and SQL tier. This would allow greater control over the workload levels and performances of the SQL VMs. As a single system is shaping the requests for both HTTP and SQL it may be able to make better dispatch systems than two separate systems could. As SQL requests are created by HTTP requests, the HTTP shaper is an oracle for the SQL shaper. The SQL shaper could then make dispatched decisions based on its current and future workload knowledge.

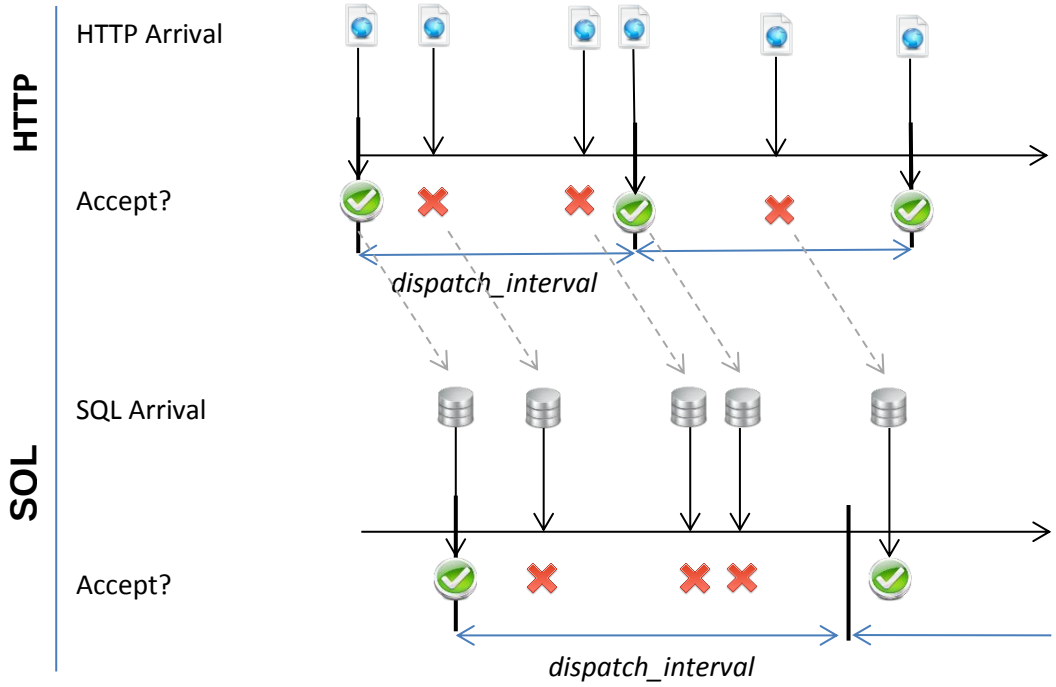


Figure 6-3: When shaping multiple technologies they may be oracles for each other

6.4.3. Multiple Resource Allocation Controllers

In our current management system we use our Dynamic Resource Controller to control the resource allocations of all VMs regardless of which data center zone they are in; shaped, over flow, or profiling. However, the Dynamic Resource Controller calculations are heavyweight operations to perform during each control period if only small changes to VMs' resource allocation levels are actually required. To mitigate this problem multiple resource controllers could be used in different zones of the data center under different conditions. For example, in the shaped zone where applications' resource allocations are almost static the control period could be increased. Alternatively, once the current Dynamic Resource Controller has allocated

the correct resource allocations a lightweight reactive controller could be used to tweak VMs' resource allocation levels as long as application-level performances are still within a certain error bound.

As VMs' workload levels in the overflow zone have large variances the resource controller could be triggered to allocate based upon workload level changes rather than using a fixed control period. Alternatively, a resource controller could utilize a hot backup system and quickly change the resources available to multiple high variance applications by editing the VMs' entries in the Input Shaper. This would allow applications' resource allocation levels to vary by multiple orders of magnitude within seconds.

6.4.4. Additional Benchmark Application Types

C-MART focuses on emulating an online bidding website. It allows clients and servers to vary their behavior, but they are still fundamentally accessing an online website. As cloud computing environments host many types of applications benchmarks that exhibit drastically different behaviors to websites are also useful for evaluating management systems. For example, a video on demand application may exhibit heavy disk and network usage, a compute engine heavy CPU and memory usage, and a scientific workload heavy GPU usage. In addition to the different resource utilization behaviors, each application may have its own performance measurement metrics such as buffering time, transactions per second, or throughput.

Although separate benchmarks could be used to emulate each different system, coordinating multiple different benchmarks and correlating the results is currently a difficult task. It would therefore be beneficial to have a single benchmark that can emulate many

different application types. This could be integrated into C-MART by having multiple client types and deploying different applications to its server VMs. However, creating such a benchmark would be a large undertaking. Also, it would require editing and updating each time a new application is added.

6.4.5. Colocation of Different Application Types

In our experiments we focus on using benchmarks that emulate interactive web applications; C-MART, RUBiS, TPC-W, and Olio. We use this application type as it is the most popular type deployed in cloud computing environments. However, cloud computing environments also contain many other application types such as data processing, data storage, video on demand, etc.

In addition to placing applications together based upon their resource requirements additional benefits can be gained by placing applications together based upon their workload types. Figure 6-4 shows an example of how placing throughput based and response time based applications together is beneficial. If there are transient increases in a response time based application's workload level it can borrow resources from a throughput based application and pay them back at a later time. This could prevent applications violating their SLOs or prevent VM migrations caused by transient overloads.

Figure 6-4 shows the performance of a video encoder and a website when co-located on the same host. After 500 seconds of the experiment the number of clients accessing the website is increased and is then decreased to its original level again at 800 seconds. It can be seen that when the resource allocations remain constant the video encoder's performance degrades by

15% and the web application's by 6000%. However, if resources are reallocated from the video encoder to the web application then the video encoder's performance degrades by 50% but the web application's performance remains constant. As video encoding is typically a long-lived process the video encoder can still achieve its SLO over the long term by increasing its video encoding rate at a later time. The website can pay back the resources it borrowed once the transient increase in its workload level has finished. Therefore, co-locating applications of different types can help reduce SLO violations.

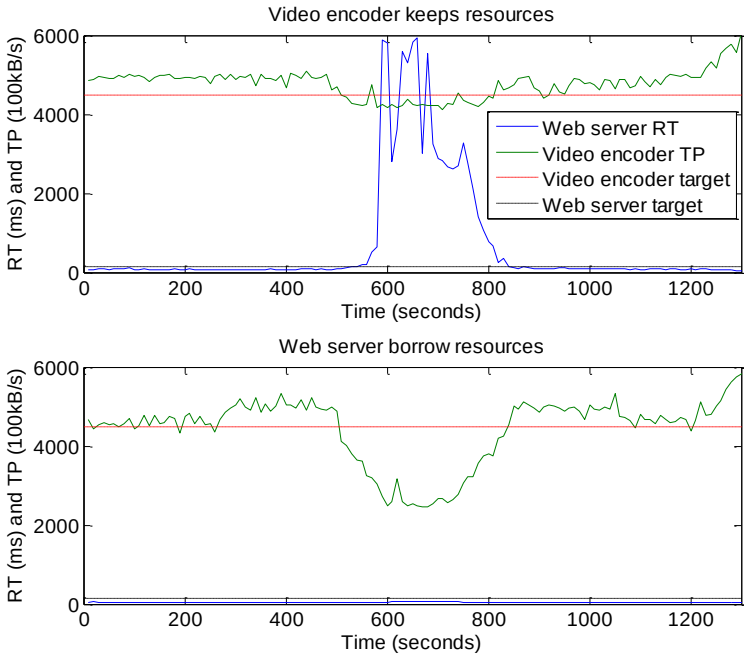


Figure 6-4: Throughput based and response time based co-location

7. Bibliography

[1] Forbes, "The Cloud Hits the Mainstream: More than Half of U.S. Businesses Now Use Cloud Computing," in *http://www.forbes.com/sites/reuvencohen/2013/04/16/the-cloud-hits-the-mainstream-more-than-half-of-u-s-businesses-now-use-cloud-computing/*, 2013.

[2] IBM, "What is cloud?," in *http://www.ibm.com/cloud-computing/us/en/what-is-cloud-computing.html*, 2013.

[3] X. Zhu, P. Padala, and Z. Wang, "Memory overbooking and dynamic control of Xen virtual machines in consolidated environments," in *IFIP/IEEE International Symposium on Integrated Network*, 2009.

[4] VMware, "VMware ESX Hypervisor," in *http://www.vmware.com/products/esxi-and-esx/overview*, 2013.

[5] Linux Foundation, "Xen Hypervisor," in *http://www.xenproject.org/developers/teams/hypervisor.html*, 2013.

[6] M. Armbrust, et al., "Above the Cloud: A Berkeley View of Cloud Computing," 2009.

[7] A. Turner, A. Sangpetch, and H. Kim, "Empirical virtual machine models for performance guarantees," in *LISA'10 Proceedings of the 24th international conference on Large installation system administration*, 2010.

[8] M. Fowler, "Patterns of Enterprise Application Architecture," in *Addison Wesley*, 2002.

[9] Amazon, "Amazon Elastic Compute Cloud (Amazon EC2)," in *http://aws.amazon.com/ec2/*, 2013.

[10] R. Kohavi and R. Longbotham, "Online Experiments: Lessons Learned," in *Computer, vol. 40, no. 9, pp. 103-105*, 2007.

[11] "TPC Benchmark W (Web Commerce) Specification," San Jose, CA, USA, 2002.

[12] "RUBiS," in *http://rubis.ow2.org/*.

[13] "Olio," in *http://incubator.apache.org/olio/*.

[14] C. Binnig, D. Kossmann, T. Kraska, and S. Loesing, "How is the weather tomorrow? Towards a Benchmark for the Cloud," in *Proceedings of the Second International Workshop on Testing Database Systems - DBTest*, 2009.

[15] B. Pugh and J. Spacco, "RUBiS revisited," in *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications - OOPSLA '04, p. 204*, 2004.

[16] Amazon, "EC2 - Amazon Web Services," in *aws.amazon.com/ec2/*, 2013.

[17] Microsoft, "Windows Azure: Microsoft's Cloud Platform," in *www.windowsazure.com*, 2013.

[18] Google, "Compute Engine — Google Cloud Platform," in *http://cloud.google.com/products/compute-engine*, 2013.

[19] D. Carrera, et. al., "Utility-based placement of dynamic Web applications with fairness goals," in *Network Operations and Management Symposium (NOMS)*, 2008.

[20] A. Karve, et. al., "Dynamic placement for clustered web applications," in *WWW*, 2006.

[21] M. Korupolu, A. Singh, and B. Bamba, "Coupled placement in modern data centers," in *IEEE Symposium on Parallel and Distributed Processing*, 2009.

[22] "VMware Infrastructure: Resource Management with VMware DRS," in *http://www.vmware.com/resources/techresources/401*.

[23] "VMware Distributed Power Management," in *http://www.vmware.com/resources/techresources/1080*.

[24] D. Gmach, et al., "Satisfying Service Level Objectices in a Self-Managing Resource Pool, Self-Adaptive and Self-Organizing Systems," in *SASO '09*, 2009.

[25] D. Gmach, et al., "Resource pool management: Reactive versus proactive or let's be friends, Computer Networks," in *Volume 53, Issue 17, Virtualized Data Centers, 3 December 2009, Pages 2905-2922*, 2009.

[26] L. Cherkasova, J. Rolia, "R-Opus: A Composite Framework for Application Performability and QoS in Shared Resource Pools," in *Dependable Systems and Networks (DSN'06)*, 2006.

[27] K. Tsakalozos, et al., "Nefeli: Hint-Based Execution of Workloads in Clouds," in *Distributed Computing Systems*, 2010.

[28] D. Williams, et al., "Overdriver: Handling memory overload in an oversubscribed cloud," in *Proc. of ACM VEE*, 2011.

[29] C. Hyser, et al., "Autonomic Virtual Machine Placement in the Data Center," in *HPL-2007-189*, 2007.

[30] F. Hermenier, et. al., "Entropy: a consolidation manager for clusters," in *ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, 2009.

[31] Choi, et.al, "Autonomous learning for efficient resource utilization of dynamic VM migration," in *International Computer Symposium*, 2008.

[32] A. Kochut, K. Beaty, "On Strategies for Dynamic Resource Management in Virtualized Server Environments," in *IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, 2007.

[33] A. Verma, P. Ahuja, A. Neogi, "pMapper: Power and Migration Cost Aware Application Placement in Virtualized Systems," in *USENIX International Middleware Conference*, 2008.

[34] A. Kochut and K. Beaty, "On Strategies for Dynamic Resource Management in Virtualized Server Environments," in *15th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, pp. 193-200*, 2007.

[35] B. Urgaonkar, P. Shenoy, and T. Roscoe, "Resource overbooking and application profiling in shared hosting platforms," in *SIGOPS Operating Systems Review, Rev. 36, SI Dec.*, 2002.

[36] R. Nathuji, et al., "Q-clouds: managing performance interference effects for QoS-aware clouds," in *EuroSys '10*, 2010.

[37] C. Stewart, et. al., "A dollar from 15 cents: cross-platform management for internet services," in *USENIX Annual Technical Conference* , 2008.

[38] X. Lui, et. al., "Optimal Multivariate Control for Differentiated Services on a Shared Hosting Platform," in *IEEE Decision and Control*, 2007.

[39] P. Padala, et. al., "daptive control of virtualized resources in utility computing environments," in *ACM SIGOPS Operating Systems Review, Volume 41 , Issue 3*, 2007.

[40] P. Padala et al., "Automated control of multiple virtualized resources," in *Proceedings of the fourth ACM European conference on Computer systems - EuroSys '09*, 2009.

[41] Swarna Mylavarapu, et al., "An optimized capacity planning approach for virtual infrastructure exhibiting stochastic workload," in *SAC '10*, 2010.

[42] Brian J. Watson, et al, "Probabilistic Performance Modeling of Virtualized Resource Allocation," in *ICAC '10*, 2010.

[43] Chen, Y., et al., "SLA Decomposition: Translating Service Level Objectives to System Level Thresholds," in *Autonomic Computing ICAC '07*, 2007.

[44] The Apache Foundation, "Apache Module mod_proxy_balancer," in *http://httpd.apache.org/docs/2.2/mod/mod_proxy_balancer.html*, 2013.

[45] A. Kamra, V. Misra, and E. Nahum, "Yaksha: A Controller for Managing the Performance of 3-Tiered Websites," in *Proceedings of the 12th IWQoS*, 2004.

[46] V. Mathur, P. Patil, V. Apte, and K. M. Moudgalya, "Adaptive admission control for Web applications with variable capacity," in *Proc. IWQOS '09, pp.1-5*, 2009.

[47] C. Huang, Y. Chusang, and C. Cheng, "Application of support vector machines to admission control for proportional differentiated services enabled Internet servers," in *Hybrid Intelligent Systems*, 2004.

[48] K. Gilly, C. Juiz, S. Alcaraz, and R. Puigjaner, "Adaptive admission control algorithm in a QoS-aware Web system," in *Modeling, Analysis & Simulation of Computer and Telecommunication Systems*, 2009.

[49] H. Zhang, G. Jiang, K. Yoshihira, H. Chen, A. Andsaxena, "Resilient workload manager: Taming bursty workload of scaling internet applications," in *In Proceedings of the 6th International Conference on Autonomic Computing and Communications, Industry Session (ICAC-INDST)*, 2009.

[50] J. Peha, and F. Tobagi, "Cost-Based Scheduling and Dropping Algorithms To Support Integrated Services," in *INFOCOM*, 1991.

[51] C. Yang, A. Wierman, S. Shakkottai, and M. Harchol-Balte, "Many flows asymptotics for SMART scheduling policies," in *IEEE Transactions on Automatic Control*, 2012.

[52] Y. Chi, H. Moon, and H. Hacıgumus, "iCBS: Incremental Cost-based Scheduling under Piecewise Linear SLAs," in *Proceedings of the VLDB Endowment*, 2011.

[53] Q. Zhang, N. Mi, A. Riska, E. Smirni, "Load unbalancing to improve performance under autocorrelated traffic," in *Proceedings of the 26th IEEE International Conference on Distributed Computing Systems, ICDCS'06*, Lisboa, Portugal, 2006.

[54] Z. Shan, C. Lin, D. Marinescu, and Y. Yang, "Modeling and performance analysis of QoS-aware load balancing of Web-server clusters," in *Journal Computer Networks: The*

International Journal of Computer and Telecommunications Networking, 2002.

[55] Hui Wang and Peter Varman, "Statistical Workload Shaping for Storage Systems," in *Proceedings of HiPC*, 2009.

[56] L. Lu, P. Varman, and K. Doshi, "Graduated qos by decomposing bursts: Don't let the tail wag your server," in *29th International Conference on Distributed Computing Systems (ICDCS)*, 2009.

[57] "eBay," in *htttp://www.ebay.com*.

[58] E. Cecchet, V. Udayabhanu, T. Wood, and P. Shenoy, "BenchLab: An Open Testbed for Realistic Benchmarking of Web Applications," in *Proceedings of 2nd USENIX Conference on Web Application Development, WebApps '11*, 2011.

[59] D.J. Abadi, M. Carey, S. Chaudguri, H. Garcia-Molina, J.M. Patel, and R. Ramakrishnan, "Cloud Databases: What's new?," in *Proceedings of the VLDB Endownment, vol. 3, no. 2-1*, Sep. 2010.

[60] Memcache, in *http://memcached.org/*.

[61] A. Beitch, B. Liu, T. Yung, R. Griffith, A. Fox, and D. A. Patterson, "Rain: A Workload Generation Toolkit for Cloud Computing Applications," in *EECS Department, University of California at Berkeley*, 2010.

[62] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proceedings of the 1st ACM symposium on Cloud computing - SoCC '10, p.143*, 2010.

[63] "Standard Performance Evaluation Corporation: SPECweb2009," in *http://www.spec.org/*.

[64] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, and R. Buyya, "CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," in *Software: Practice and Experience, vol. 41, no. 1, pp. 23-50*, Jan. 2011.

[65] Apache Lucene - Apache Solr, in *http://lucene.apache.org/solr/*.

[66] MongoDB, in *http://www.mongodb.org/*.

[67] The Apache Cassandra Project, in *http://cassandra.apache.org/*.

[68] C-MART, in *http://theone.ece.cmu.edu/cmart/*.

[69] P. Bodik, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson, "Characterizing, modeling, and generating workload spikes for stateful services," in *Proceedings of the 1st ACM symposium on Cloud computing - SoCC '10, pp. 241-252.*, 2010.

[70] Apache Software Foundation, "Apache HTTP Server," in *http://httpd.apache.org/*, 2013.

[71] The Apache Software Foundation, "Apache Tomcat," in *tomcat.apache.org*, 2013.

[72] The Apache Software Foundation, "The Apache Cassandra Project," in *cassandra.apache.org*, 2013.

[73] www.netcraft.com, "June 2013 Web Server Survey," in *http://news.netcraft.com/archives/2013/06/06/june-2013-web-server-survey-3.html*, 2013.

[74] www.builtwith.com, "jQuery Usage Statistics," in *http://trends.builtwith.com/javascript/jQuery*, 2013.

[75] X. Zhu, P. Padala, and Z. Wang, "Memory overbooking and dynamic control of Xen virtual machines in consolidated environments," in *IFIP/IEEE International Symposium on Integrated Network Management, pp. 630-637*, 2009.

[76] C. Stewart, T. Kelly, and A. Zhang, "Exploiting nonstationarity for performance prediction," in *ACM SIGOPS Operating Systems Review - EuroSys'07 Conference Proceedings*, 2007.

[77] X. Huang, W. Wang, W. Zhang, J. Wei, and T. Huang, "An Adaptive Performance Modeling Approach to Performance Profiling of Multi-service Web Applications," in *IEEE 35th Annual Computer Software and Applications Conference*, 2011.

[78] A. Downey., "Lognormal and Pareto distributions in the Internet," in *www.allendowney.com/research/longtail/downey03lognormal.pdf*, 2003.

[79] P. Barford, A. Bestavros, A. Bradley and M.E. Crovella, "Changes in Web client access patterns: characteristics and caching implications," in *World Wide Web, Special Issue on Characterization and Performance Evaluation*, 1999.

[80] Stephen P. Boyd, "Information Systems Laboratory, Department of Electrical Engineering, Stanford University," in *http://www.stanford.edu/~boyd/software.html*, 2013.

[81] Google App Engine, "Backend Java API Overview," in *http://code.google.com/appengine/docs/java/backends/overview.html*, 2013.

[82] mod_form, in *http://apache.webthing.com/mod_form/*, 2013.