CORE
Metadata, citation and similar papers at core.ac.uk
Provided by Drexel Libraries E-Repository and Archives

A Parallel Controller Implementation for Dynamic Resource Allocation in Virtualized Computing Environment

By

Shahab Ahmad

A thesis submitted in partial fulfillment
of the requirement for the degree of

Masters of Science in Computer Engineering

at

Drexel Univeristy

July 2008

© Copyright 2008
Shahab Ahmad. All Rights Reserved.

Acknowledgments

The author wishes to express sincere appreciation to Dr. Nagarajan Kandasamy and Dara Kusic for their assistance in the preparation of this manuscript. Without their insight, patience, and guidance this thesis would not have been possible. I would also like to extend my gratitude to Dr. Baris Taskin and Dr. Harish Sethu for their wisdom and guidance.

Additionally, I would like to specially thank my parents, my brothers and my sister for their help and support.

Table of Contents

CHAPTER 1 INTRODUCTION 2

1.1 BACKGROUND 3
1.2 PROBLEM STATEMENT 5

CHAPTER 2 PRELIMINARIES 6

2.1 SYSTEM COMPONENTS: 6
2.2 APPLICATION CLUSTERS: 10
2.3 WORKLOAD CHARACTERISTICS 13
2.4 ONLINE CONTROL CONCEPTS 16
2.5 LIMITATIONS OF LLC 17

CHAPTER 3 PROPOSED SOLUTION 20

3.1 MAIN DESIGN OBJECTIVES 20
REDUCE POWER CONSUMPTION: 20
MINIMIZE NEEDED RESOURCES: 20
MAXIMIZE SERVER PERFORMANCE: 21
MANAGE LARGE NUMBER OF MACHINES EFFICIENTLY: 21
FLEXIBLE APPLICATION DESIGN: 21
MINIMIZE SLA VIOLATIONS: 22
3.2 OBJECT DESIGN OF DYNAMIC RESOURCE PROVISIONING FRAMEWORK 22
LLC OBJECT: 23
MACHINE OBJECT: 23
PHYSICAL MACHINE OBJECT: 23
VIRTUAL MACHINE CLASS: 24
APPLICATION CLUSTER OBJECT: 25
VM STATE TABLE: 26
PM STATE TABLE: 27

Frequency State Table: 27
State Search Tree Object: 29
System Configuration Class: 30
3.3 DRPF Behavioral Design: 31
Program Startup: 31
Initialization Machines and Hosts: 31
Creating Frequency State Table: 32
Creating VM State Table: 33
Creating State Search Tree: 34
Loading Incoming and Predicted Workload Files: 35
Updating Physical Machine States: 36
Updating Virtual Machine States: 36
Obtaining System Dynamics: 36
3.4 Controller Design and Implementation 38

CHAPTER 4 SIMULATION RESULTS 43

4.1 Control Performance 43
4.2 Execution Speedup via Parallelization: 46

CHAPTER 5 CONCLUSIONS 49

BIBLIOGRAPHY 50

List of Figures

Figure 1: A heterogeneous cluster of servers consisting of different processors, memory and cache configuration 4
Figure 2: Virtual Machine and Physical Machine state transition diagram 10
Figure 3: System Model consisting of Gold and Silver Application Clusters with both powered ON and powered OFF machines 11
Figure 4: Gold Application Processing Rate 12
Figure 5: Silver Application Processing Rate 12
Figure 6: The Incoming Workload for both Gold and Silver Applications 14
Figure 7: Actual and Predicted Workload for Gold Application 15
Figure 8: Actual and Predicted Workload for Silver Application 15
Figure 9: The exponential Increase in the number of VM States as the number of servers is increased 19
Figure 10: The Class Diagram of Machine, Physical Machine and Virtual Machine showing their members 24
Figure 11: The Class Diagram of Cluster Class that represents client applications 25
Figure 13: A small portion of a VM State Table that consists of all possible ways of assigning 6 VMs to Gold Application and 1 VM to Silver Application. 26
Figure 14: The states of 7 PM over 5 time steps showing machines turning ON and OFF 27
Figure 15: A 4-Dimensional Frequency State Table with four PMs 28
Figure 16: A 2-Dimensional Frequency State index table 28
Figure 17: State Search Tree Object showing span of tree 29
Figure 18: The Class diagram of State Object 30
Figure 19: The class diagram of ConfigFile Class 31
Figure 20: VM State Table application VM assignments 33
Figure 21: An example of VM State Table that consists of all possible ways of assigning VMs to client applications 34
Figure 22: The Algorithm used to create the State Search Tree 35
Figure 23: The Algorithm used to compute cost 37
Figure 24: An outline of the parallelized prediction algorithm 38
Figure 25: The Algorithm used to compute the minimum number of VMs needed for gold and silver applications 40
Figure 26: An example showing how multiple threads explore different VMs states in parallel 40
Figure 27: The Algorithm used to limit the number of states explored for prediction purposes 41

Figure 28: An example showing how the best state is chosen among many alternatives 42
Figure 29: Power Consumption comparison of controlled and uncontrolled systems 44
Figure 30: Incoming Workload for both gold and silver application 44
Figure 31: Number of Physical Machines ON and OFF during 24 hours period 45
Figure 32: Number of Virtual Machines ON and OFF during 24 Hours Period 46
Figure 33: The Average Prediction Time of the controller VS. number of threads 47
Figure 34: Execution Speedup obtained via parallelization of the prediction algorithm 48

Abstract

A Parallel Controller Implementation for Dynamic Resource Allocation in Virtualized Computing Environment

Shahab Ahmad
Advisor: Nagarajan Kandasamy, Ph. D.

The ability to dynamically allocate system resources in a large scale distributed system is highly desirable. Dynamically allocating system resources can significantly reduce under-utilization of system resources and reduce the power consumed by the servers. Since typical enterprise computing systems consist of hundreds of servers, it is almost impossible to manually reconfigure each system parameter for optimal performance. Prior work has shown that by posing the dynamic resource provisioning problem as one of sequential optimization, we can dynamically allocate system resources for optimal performance in a dynamic operating environment. However, a single threaded implementation of this control technique does not scale well with increasing system size. Therefore, this thesis develops a parallel controller implementation for dynamic resource allocation using the OpenMP interface. We analyze the performance of this controller in a virtualized computing environment, and show that dynamic resource allocation can lead to an average of 30% savings in energy consumption, over an uncontrolled system. Parallelizing the controller also significantly reduces its execution time overhead, by as much as 263%, a compared to single threaded implementation.

Chapter 1
Introduction

On-demand computing, a provisioning model where a computer service provider makes system resources available to clients as needed is becoming increasingly common in enterprise computing systems. The computing infrastructure typically consists of heterogeneous cluster servers that can service multiple clients simultaneously. Companies typically overprovision system resources to handle the highest possible incoming workload. For online banking and shopping applications that consists of many replicated servers running the application, server utilization is usually less than 20% by many estimates.

The emergence of virtualization technology in recent years has enabled companies to consolidate their data centers and reduce power consumption. In a virtualized environment, a single host can be transformed into multiple virtual servers that share system resources. Each virtual machine is a performance-isolated platform that provides the same functionality as a physical machine. Dynamically provisioning virtual machines by turning both physical servers and virtual servers ON and OFF allows service providers to consolidate the workload on a smaller number of virtual machines. The service providers can conserve power and improve system utilization while maintaining desired quality-of-service (QoS).

Dynamically allocating system resources require tuning the system for optimal performance for incoming workload. To adjust the system's configuration, we need to accurately predict the behavior of the system under incoming workload. For complex distributed systems, predicting the future behavior requires exploring millions of possible state configurations and choosing the optimal state as the next state of the system. Exploring each of these states requires realizing the system's behavior as each of these states is applied to the system. Therefore, exploring millions of states can be very computation intensive.

This paper details an optimization approach that takes advantage of parallelization techniques to quickly and efficiently solve the dynamic resource allocation problem. It uses the Limited Lookahead Controller (LLC), previously developed in [1], [2], to predict the future state of the system. To improve the performance of LLC, our application uses OpenMP to parallelize the state search process. This approach considers all feasible configurations of the system and then uses OpenMP to explore these states in parallel. This paper also discusses the optimization techniques used to further enhance the performance of the dynamic resource provisioning model.

1.1 Background

On-Demand resource allocation is an emerging resource provisioning model that makes computing infrastructure available to customers as needed, and realizing this model requires making dynamic decisions about the future state of a distributed system. A distributed system typically consists of hundreds of servers that are connected together by a network. These large systems are typical for many fortune 500 companies. For example, Google, a search industry giant, currently utilizes more than 200,000 serves across the globe to provide search capabilities to its users. Since each individual server normally consist of many tunable parameters, tuning many servers for optimal performance becomes very difficult and cumbersome.

In a typical enterprise environment, the distributed systems normally consist of a heterogeneous cluster of servers. Each server may be from a different vendor, with different processor and memory configurations. Furthermore, each server can run a different operating system. Therefore, the configurable parameters may differ greatly from one server to another. Having to configure hundreds of servers is difficult enough, but having to configure hundreds of different servers for optimal performance is even more difficult. Figure 1 below shows an example of a heterogeneous set of serves.

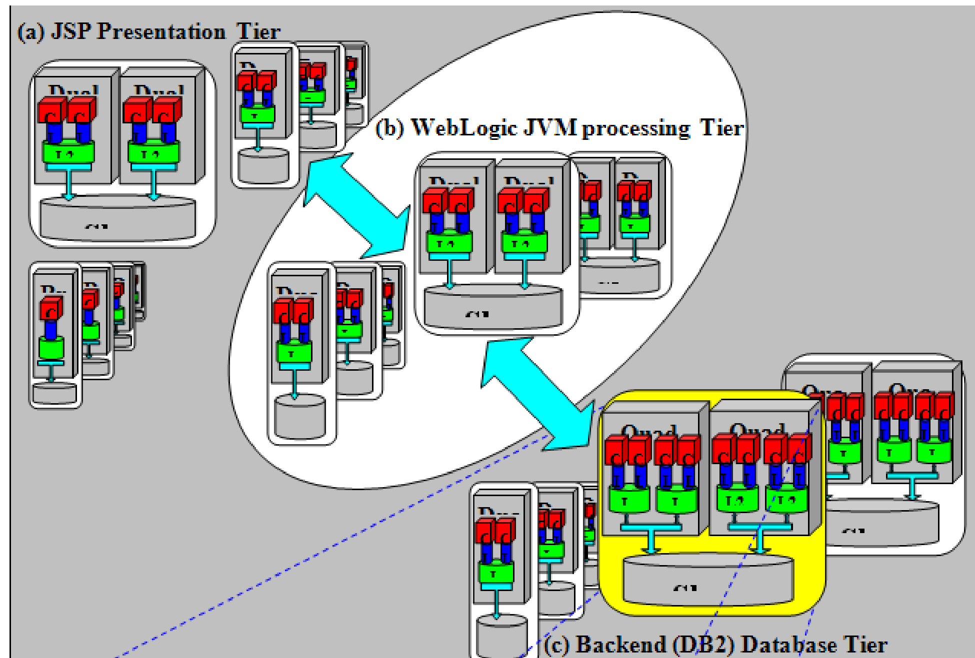

Figure 1: A heterogeneous cluster of servers consisting of different processors, memory and cache configuration

The introduction of virtualization technology has revolutionized the enterprise computer industry. Virtualization allows companies to host multiple virtualized servers on a single physical server. The virtualization technology is enabling companies to further extend the capabilities of individual physical servers by dividing them in multiple virtual servers. Since each virtual server can be configured independently, the number of different configurations is increased even further. Therefore, the ability to virtualize servers further complicates the configurations and makes it even more difficult to tune the distributed system for optimal performance.

Enterprise level distributed systems can host anywhere from one to many applications simultaneously. When a distributed system is hosting multiple applications, it must share resources among the different applications. These applications may run independent of each other and their workload intensity may vary greatly over time. So the system resources required by an application vary over time. For online applications that experience time varying workload, it is especially true. For such

applications, a system administrator does not only need to configure system parameters for optimal performance initially, but he also needs to continuously reconfigure system parameters to maximize system performance.

1.2 Problem Statement

While a distributed system consists of hundreds of configurations and many tunable parameters, the parameters consists of a finite set of possibilities. Therefore, while it may be cumbersome to manually tune multiple parameters for an optimal configuration, we can programmatically tune these parameters for optimal configuration. By posing the dynamic resource provisioning problem as a set of sequential optimizations under uncertainty, the distributed system resources can be dynamically allocated for optimal system performance. The sequential optimization problem can be solved using a Limited Lookahead Control (LLC) scheme.

Having a large number of idle servers in a cluster leads to higher power consumption. Due to the increasing cost of energy in the past decade, it is becoming more important to reduce power consumption to minimize cost of operations. By dynamically switching machines ON and OFF depending on the workload requirements, service providers can reduce their cost.

Solving the resource allocation problem millions to times requires extensive computational power and time. To maximize the benefits of dynamically changing system state, the controller must be able to quickly predict and apply the new system state. Therefore, it’s imperative to be able to quickly predict and apply a new system state that can improve system performance and reduce its cost.

Chapter 2
Preliminaries

2.1 System Components:

Our system model consists of a heterogeneous cluster of Dell PowerEdge serves containing PowerEdge 1950 and 2950 series. Table 1 below shows the frequency and processing core configurations for each server. Our actual System model configuration consists of 12 physical servers consisting of PowerEdge 2950 and 1950 servers. At any given time, each server may be either in on, off, booting, or in shutting down state.

To minimize the system resources needed to process the workload, our model actively shuts down or boots up the physical servers. The machines are turned ON and OFF based on the workload requirements. A physical server at any given time may be in one of five states, and the state is determined based on the processing needs. Table 2 below shows the different possible state and acceptable transitions for the physical servers. Through experimental tests, we have learned that it takes about four minutes to shut down or boot up a physical host. Since our controller comes up after every 120 seconds (1 time step), shutting down or booting up a host takes two time steps.

It's important to note that shutting down a host is highly risky because when a host is given the order to shut down, it will become off-line for four time steps (2 time steps to turn off and 2 time steps to turn on). During this time, if the workload intensity increases and that specific physical machine is needed to process the incoming requests, our system will not be able to respond to the incoming requests. This will lead to queuing in the system and if the request are not handled promptly, they will lead to SLA violations. Therefore, extreme care must to be taken to minimize the number of SLA violations.

Types of Physical Machines in System Model				
Server Name	**Number of Processors**	**Cores per Processor**	**Core Frequency**	**Frequency Sum**
PowerEdge 2950	2	4	2250 MHz	2*4*2250=18GHz
PowerEdge 1950	2	4	1500 MHz	2*4*1500=12GHz

Table 1: The Types of Physical Servers in the System Model

Physical Machine State Transition and Constraints				
State Description	**State ID**	**2950 Power Consumption (watts)**	**1950 Power Consumption (watts)**	**Acceptable State Transition***
Powered Off	-2	0	0	-2→ 1
Shutting Down	-1	241.0	207.0	-1 → -2
Standby	0	19.5	18.5	0 → 1
Booting Up	1	295.5	250.0	1 → 2
Powered On	2	241+ 45*VMs	207 + 45*VMs	2 → -1 2 → 0

Table 2: The State Transition of the Servers

Our system model takes advantage of the virtualization technology and host different client applications within multiple virtual servers. Each Dell PowerEdge server hosts multiple virtual servers to handle incoming workload. Table 3 below shows the virtual machine configuration and constraints. The PowerEdge 1950 sever can host a maximum of 3 virtual machines and the PowerEdge 2950 can host a maximum of 4 virtual machines. Since a virtual machine needs at least 2 GHz to support its operating system and application server processing needs, our model puts a 2 GHz minimum frequency constraint on a virtual server. Our system model consists of 44 virtual machines mounted on 12 physical servers.

Virtual Machine Configuration and Constraints				
Server Name	**Max VMs**	**Max Frequency Sum**	**Max VM Frequency**	**Min VM Frequency**
PowerEdge 2950	4	18 GHz	9000 MHz	2000 MHz
PowerEdge 1950	3	12 GHz	6000 MHz	2000 MHz

Table 3: The Types of Physical Servers in the System Model.

We used VMware ESX server to create virtual machines. One of the inherent constraints that VMware places on virtual servers is that a Virtual Machine (VM) cannot access more than four cores on a physical host. Since we use two quad core machines on our servers, we effectively have eight addressable cores on each physical host. VMware's constraints limit the maximum frequency that a virtual machine can attain. Therefore, A PowerEdge 2950 server can only assign a maximum of 9 GHz to its VM (4*2250 = 9000 MHz).

To limit the set of possible frequency configurations, we place an implicit constraint on the VM frequency increments. A VM can attain any frequency between its minimum and maximum frequency range with 1000 MHz increments. For example, a VM mounted on PowerEdge 1950 may select a frequency from the set {2000, 3000, 4000, 5000, and 6000}. Placing this constraint allows us to limit the size of our frequency configuration search space.

At any given time, a physical server in our cluster can host multiple VMs simultaneously. When multiple VMs are running on a single server, they must share common resources such as the CPU. Therefore, the sum of the virtual machines frequencies cannot exceed the total frequency of the physical host. For example, if a PowerEdge 1950 server is hosting 3 VMs, then a frequency configuration of 6000, 3000, and 3000 for the three VMs is valid. However, a frequency configuration of 6000, 4000, and 6000 is not valid because the sum of the frequencies exceeds 12000 MHz. This hardware restriction is especially useful in limiting the frequency state space because it provides an upper bound.

In a virtualized environment, it is beneficial to allocate the maximum possible frequencies to each host because using the minimum frequency to process the incoming workload has very minimal effect on the power savings [2]. Therefore, in our model, the sum of the frequencies on a physical host always adds up to the maximum frequency that a host can support. For example, if a host can support a maximum of 12000 MHz, and it currently have three operating VMs. If the three VMs need 4000, 2000, and 3000 (sum=9000) MHz respectively to process the workload, our model will assign a combination of 4000, 4000, and 4000 (sum=12000) MHz respectively to each VM. Imposing this implicit constraint enables us to further reduce the size of frequency state space.

To minimize the system resources needed to process the workload, our model actively shuts down or boots up the virtual severs. If the incoming workload is low and a virtual machine is not needed to processing any work, the system model will give the signal to turn off the virtual machine. In our model, it takes one time step to turn off a virtual machine. Turning off a virtual machine saves power and reduces cost. Therefore, it is desirable to turn off virtual machines when they are not needed. Note that since one physical server may host up to three or four virtual servers, turning off a virtual machine does not necessarily mean that we will also need to turn off the physical host. A host if turned off only when all of its VM's are turned off and our system model predicts that none of the VMs hosted on the PM will be needed. Figure 2 below shows the state transition table.

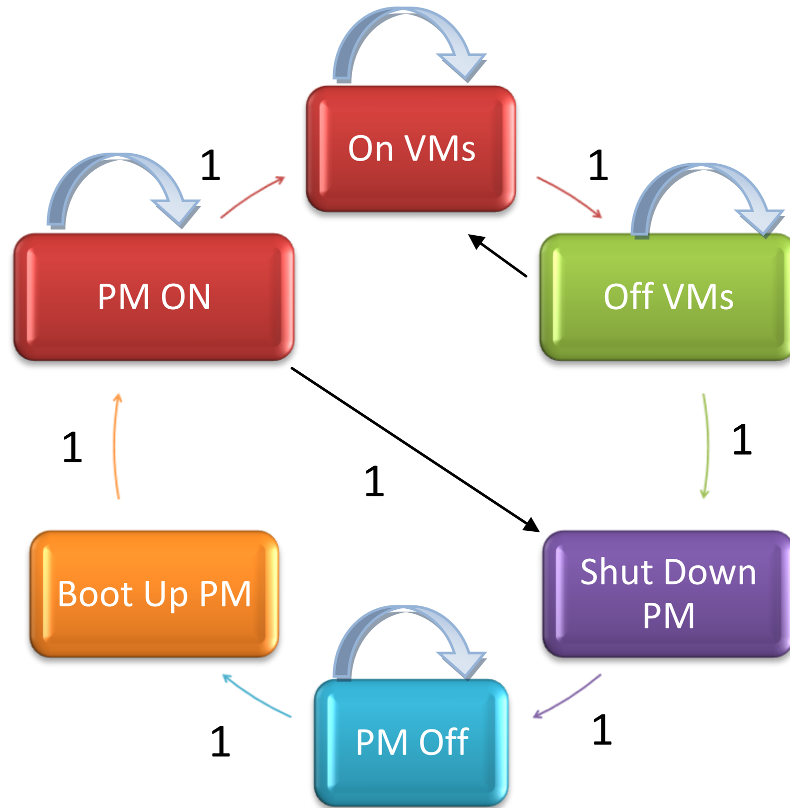

Figure 2: Virtual Machine and Physical Machine state transition diagram

2.2 Application Clusters:

Our system model uses two independent online applications labeled as "Gold" and "Silver" where each application is indexed using $i \in \{1, 2\}$. $\lambda_1(t)$ and $\lambda_2(t)$ represents the arriving workload at time t for both Gold and Silver application clusters, respectively. As shown in figure 3, the incoming workload is routed to the appropriate cluster via a global dispatcher. Each application cluster itself consists of a WL dispatcher. The Application Workload (WL) dispatcher works as a load balancer and divides the arriving workload among active virtual machines. Each virtual machine receives a fraction of the workload γ based on its processing capabilities. The amount of workload given to an individual virtual server also depends on the current queue of the virtual server. If a server currently has an existing queue, then the dispatcher scales back the workload share γ given to a virtual machine.

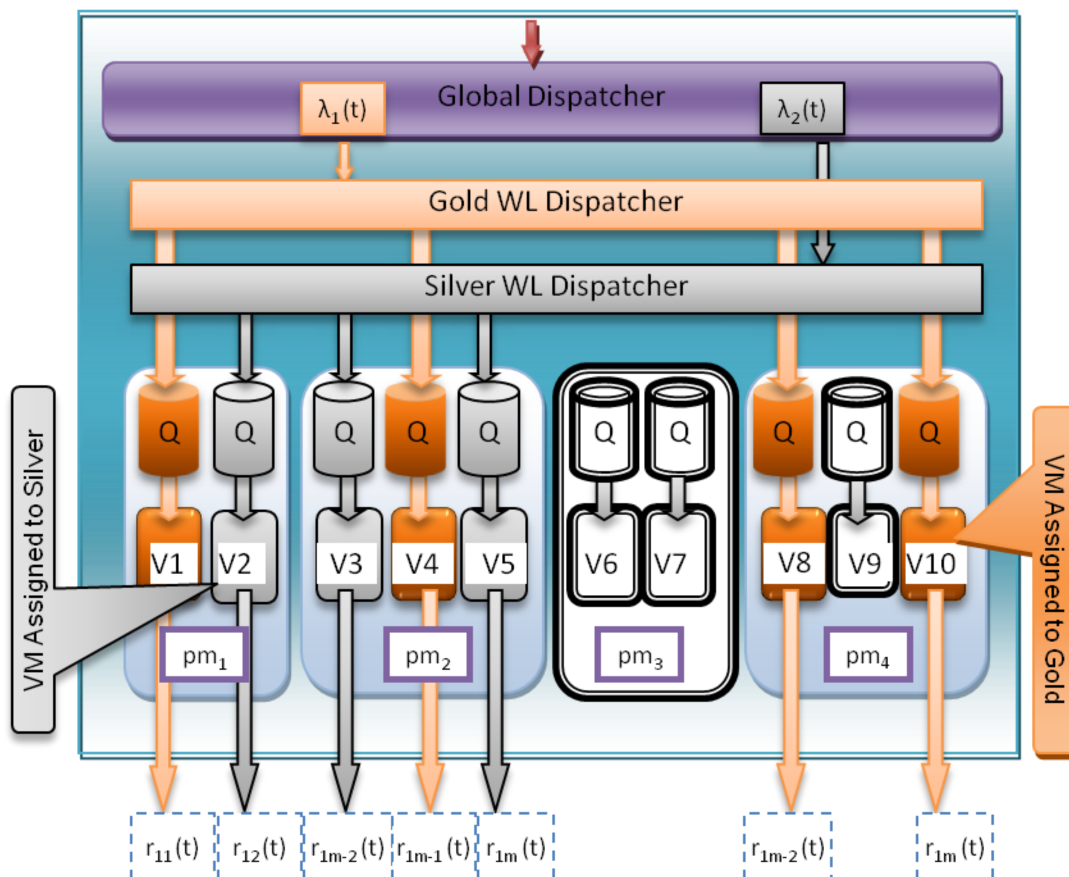

Figure 3: System Model consisting of Gold and Silver Application Clusters with both powered ON and powered OFF machines

Typical distributed systems consist of multiple application clusters processing requests at different rates. In our system model, the Gold and Silver applications process the arriving workload at different rates. While these two applications may share system resources, an individual applications processing rate dependent solely on the available frequency and an existing work queue. Figure 4 & 5 below shows the processing rate for the tow applications with and without the queue.

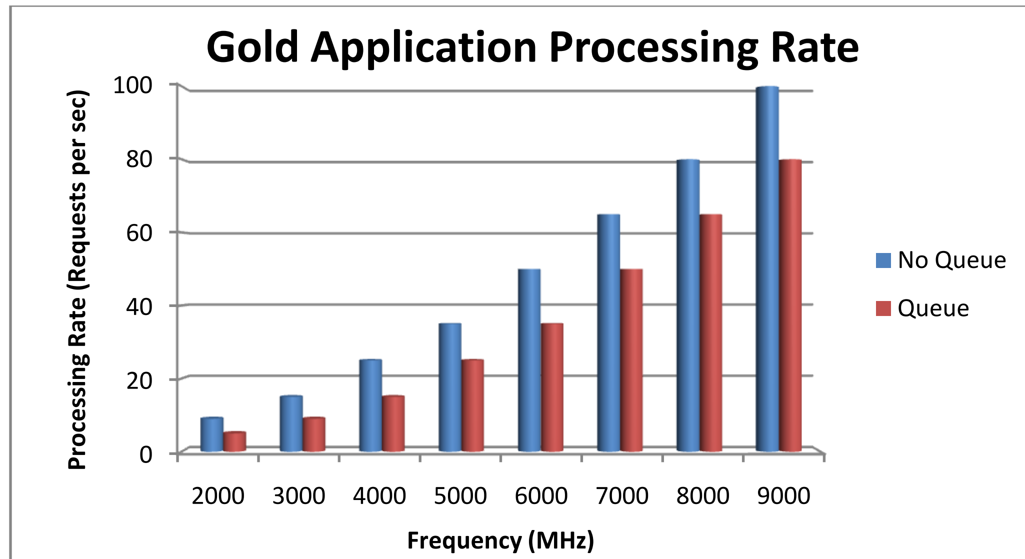

Figure 4: Gold Application Processing Rate

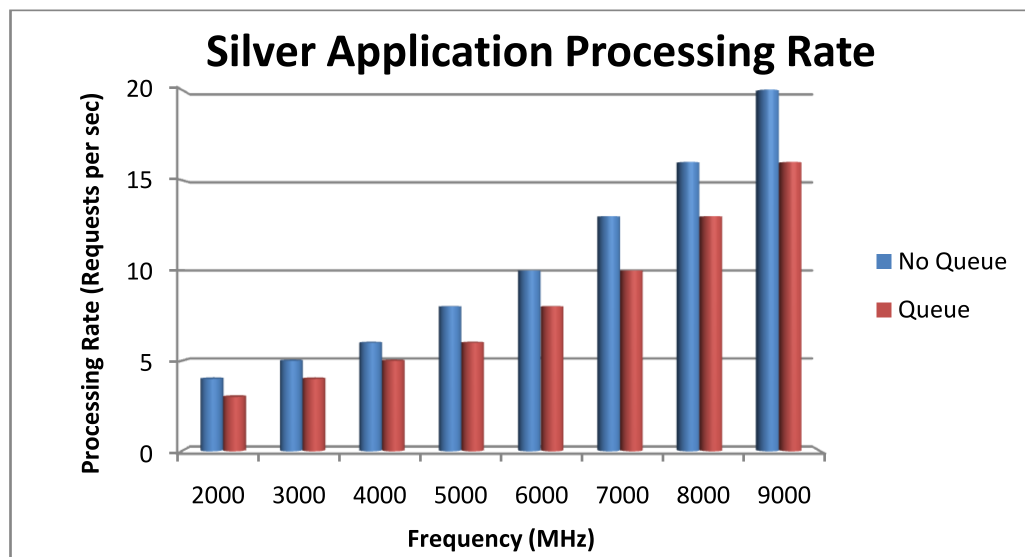

Figure 5: Silver Application Processing Rate

The two applications in our system share 44 virtual servers mounted on 12 physical servers. At any given time, each application has at least 1 virtual machine dedicated towards processing the incoming request. As the workload intensity increases, the number of virtual machines assigned to each application also increases. Depending on the processing

requirements, a virtual machines mounted on a single physical machine may be serving one or both applications simultaneously. For example, at run time, a PowerEdge 1950 server may be hosting two virtual servers serving gold and silver applications with each running at 6000 MHz (third VM is off). When number of VMs needed to process the workload is greater than the maximum number of VMs, then priority is given Gold application because it is the higher paying client.

Our system consists of 3 tunable parameters: (1) the number of virtual machines assigned to each application; (2) the number of physical machines to turn on or off; (3) the CPU share assigned to each VM to process the workload. Each tunable parameter consists of a finite set of states to choose from. To maximize utility, our goal is to choose an optimal configuration from the tunable parameters.

For any given incoming workload, our goal is to use the minimum number of virtual machines and physical machines to process the workload. To reduce power consumed by physical hosts, our controller tries to bunch together virtual machines on as few physical machines as possible. There is however a disadvantage of consolidating the VMs on the fewest possible number of physical hosts; As the number of VMs hosted on a PM increases, the average CPU share of the VMs also decreases. This leads to a decrease in the throughput. To maximize the throughput, our model constructs a CPU frequency state table and searches for an optimal configuration that can adequately handle the incoming workload.

2.3 Workload Characteristics

Each application receives a time varying workload in terms of requests per second. The workload files were obtained from 1998 World Cup Soccer web site's. These files include the incoming workload requests during a 24-hour period in 120 second intervals. Figure 6 below shows the incoming request for both Gold and Silver application over a 24 hour period. Note that this specific workload includes the worst case scenario where both of the applications workload peaks at the same time.

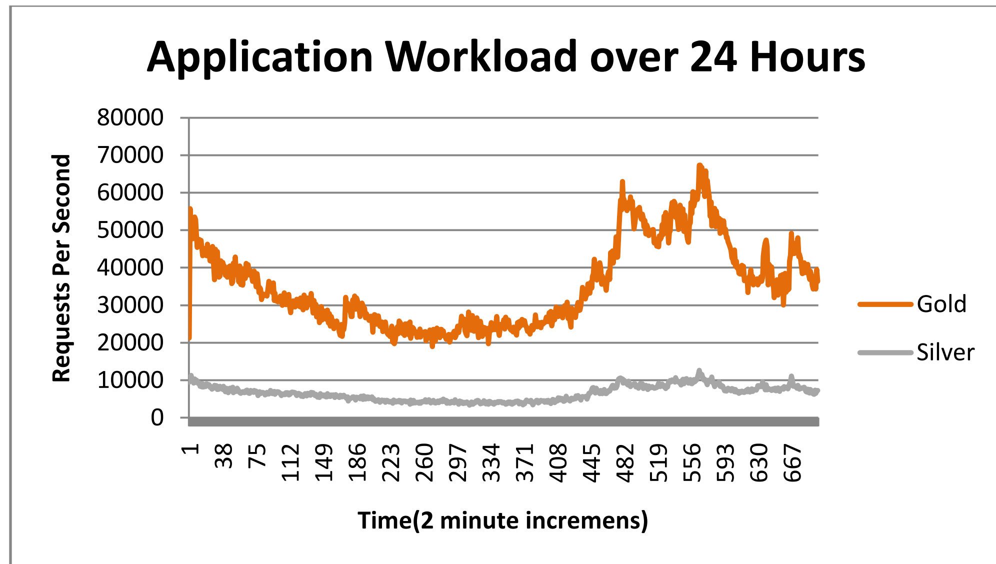

Figure 6: The Incoming Workload for both Gold and Silver Applications

To predict the future behavior of the applications, our system model uses a predicted workload. The predicted workload is generated using Kalman filter. The filter is first trained using a small portion of the actual workload, and then used to forecast the remainder of the load during controller execution. Figure 7 and 8 below shows the difference between the actual incoming requests and the predicted requests. It's important to note that the predictor generally over-predicts to reduce the number of SLA violations. The over prediction helps avoid under allocation of system resources which may lead to queuing. The Kalman filter does a fairly good job in minimizing the prediction error. The gold application experiences a 6% prediction error while the silver application experiences a 7% prediction error.

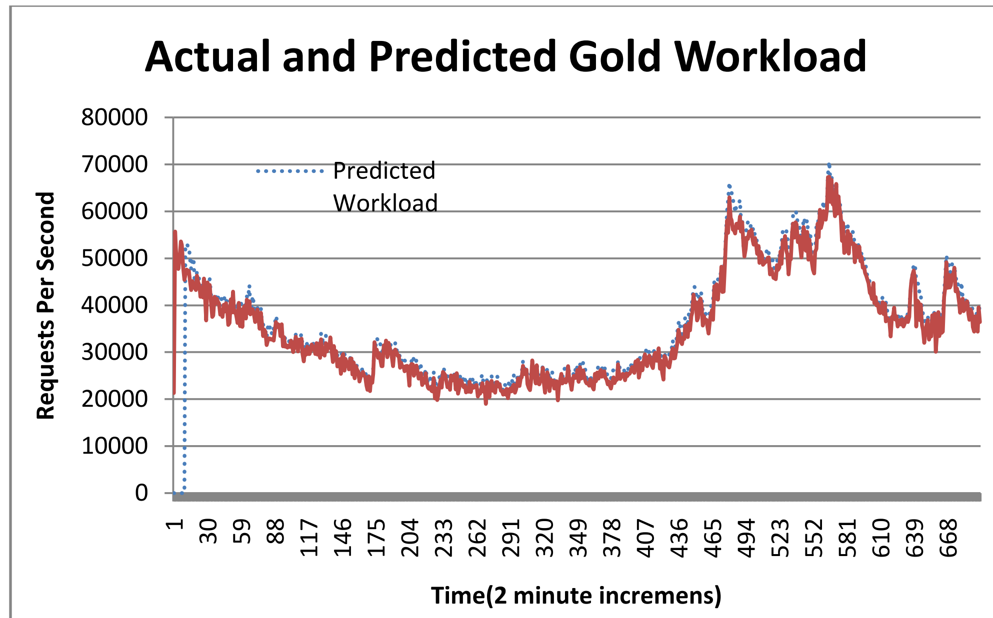

Figure 7: Actual and Predicted Workload for Gold Application

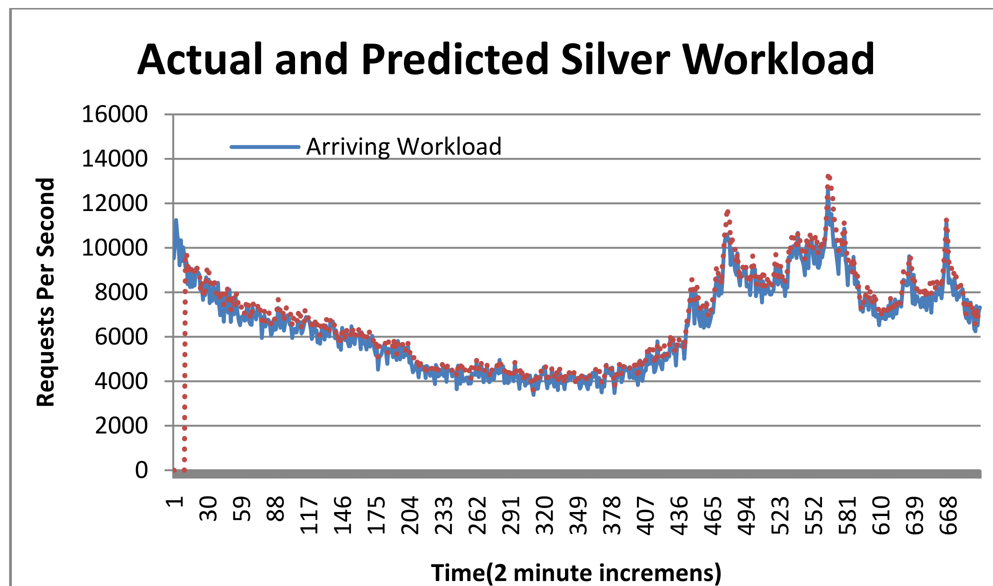

Figure 8: Actual and Predicted Workload for Silver Application

Among the main objectives of our system model is to minimize the power consumed by a distributed system. To achieve this goal, our model actively uses the minimum possible system resources to process the

incoming workload. To minimize the number of resources used, the LLC controller predicts the minimum number of VMs needed to process the incoming workload. Any virtual or physical hosts that are not needed to process the workload are shut down to conserve power.

2.4 Online Control Concepts

Limited Lookahead Controller (LLC) framework is an online control framework for self-managing distributed computing systems. This framework continuously configures and reconfigures the distributed systems for optimal performance in response to changing computing demands and environmental conditions [1]. This framework uses a model-predictive control approach in discrete domain where the control actions optimizing the distributed system's Quality of Service (QoS) are derived over a limited prediction horizon. At each time step over the prediction horizon, the optimal control actions are obtained by solving a multi-variable (finite) objective function. This objective function specifies the trade-offs between achieving the desired QoS and the corresponding cost incurred in terms of resource usage.

The objective function estimates the system behavior by generating a discrete-time state-space equation

$$x(t+1) = \phi(\, x(t), u(t))$$

where x(t) and u(t) $\in \{u_1, u_2, \ldots, u_r\}$ denote the sampled form of the continuous state vector and the discrete valued input vector at time t, respectively. The state vector consists of all possible state configurations (finite set) for a specific distributed system. To achieve the desired QoS objectives, the algorithm uses Utility optimization performance specifications. The utility optimization is used to maximize (or minimize) a given performance measure represented as a function of state and input variables.

To predict the system's state at next time step $\mathrm{x}(\mathrm{t}+1)$, the LLC framework explores a limited look-ahead horizon within the system state space and selects the next state that provides the maximum utility. The

system state space is expanded into a tree and each state is given a utility function. This function assigns a cost (revenue) associated with reaching and maintaining a specific state. In exploring this tree, the LLC framework searches for the set of states that best satisfy the QoS specifications. A state x_m that provides the minimum cost is selected from the finite list of states. The selected state is tracked back to the current state and the input leading to x_m is chosen as the given to the discrete-time state-space function.

To estimate the system's behavior over the prediction horizon, relevant environment parameters such as the arriving workload and per request processing rate must first be predicted. For Web Server workloads, the arriving requests (workload) can be predicted using a Kalman filter[3]. The average request processing time is estimated using an exponentially-weighted moving average (EWMA) filter as [4]

$$\hat{u}(t) = \pi \times u\,(k) + (1 - \pi) \times u(t-1)$$

where π is a smoothing constant.

The model predictive control approach is especially effective for switching hybrid systems where the set of possible control inputs are finite. Switching hybrid systems are practical distributed systems that have a finite set of possible control inputs, and they exhibit hybrid behavior comprising of both discrete-event and time-based dynamics.

2.5 Limitations of LLC

Typical distributed systems consist of large cluster of servers. The ultimate goal of dynamic resource provisioning frameworks is to be able to manage hundreds of servers. However, the number of manageable serves is limited by the computation power and the memory available on the servers.

The exponential growth of the system state search space limits the maximum number of manageable servers. As the number of servers to be managed grows linearly, the number of possible system states grows exponentially. Predicting the future state of a large cluster of servers

involve iterating through millions of possible states. As the prediction algorithm iterate over each individual state, it considers the cost and revenue associated with attaining this state. Additionally, the controller also considers the risk associated with turning on or off individual servers (virtual servers and physical servers).

Computing the cost, revenue and the risk associated with each state millions of times as the prediction algorithm iterate over all possible configurations require extensive computational power. Therefore, the ability to manage large number of serves is limited by the computation power of available servers. Figure 9 below shows the growth in the search space as the number of servers is increased.

As the number of system states grow, the amount of memory needed to store the states grow also. Each individual state not only stores the virtual machine configuration, but it also stores frequency state configuration and physical machine state configuration. The available memory is further strained by the need to store the state information over multiple horizon steps. Therefore, an exponential growth in the number of stats leads to an exponential growth in the amount of memory needed to store the states. The amount of memory available on the machine running the dynamic resource provisioning framework application limits the maximum number of servers that can be managed.

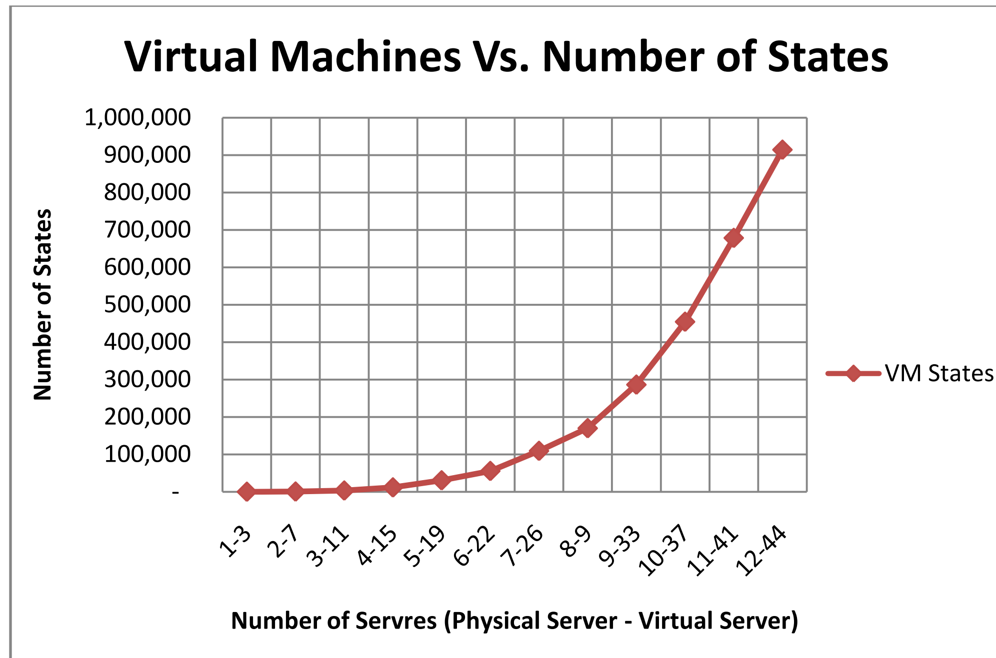

Figure 9: The exponential Increase in the number of VM States as the number of servers is increased

Chapter 3
Proposed Solution

3.1 Main design objectives

The main objective is to design a high performance dynamic resource provisioning framework (DRPF) that can reduce power consumption and under-utilization of distributed systems. The DRPF must be able to maximize system throughput via dynamically configuring system parameters for optimal performance while managing a large number of servers. The framework should also be aware of the risks associated with sudden changes in the workload and be able to adapt to workload fluctuations while minimizing SLA violations.

Reduce Power Consumption: In typical data centers today, server utilization usually averages about 6%. Furthermore, up to 30% of the servers are typically idle. The significant underutilization leads to higher power consumption and waste of system resources. Our main design objective is to minimize the power consumption of large scale distributed systems. Using the dynamic resource provisioning framework discussed in this paper, we can significantly reduce the power consumption by dynamically allocating the minimum number of resources needed to handle incoming workload.

Minimize Needed Resources: In Most service providers overprovision system resources to adequately handle the highest possible workload specified in the Service Level Agreement (SLA) by a client. For online e-commerce applications that experience a time varying workload, the workload intensity very rarely reaches its maximum level. During intervals of low workload intensity, the incoming requests can ideally be processed with minimal system resources. Minimizing the resources needed by a specific application allows service providers to provision the resources to additional clients and maximize their return on investment. It is highly beneficial to use only the resources necessary to process client workloads. Therefore, our objective is to design a framework that utilizes minimal system resources.

The resource provisioning framework discussed in this paper reduces the number of system resources needed to process client requests. At any given time, our framework dynamically provisions virtual servers to client applications based on the processing needs.

Maximize Server Performance: To use minimum system resources to process the workload, a dynamic resource allocation framework must be able to optimally configure the distributed system for best performance. The framework must be able to continuously configure and reconfigure the distributed system to reflect the changes in the workload. The framework proposed in this paper actively configures the CPU share of virtual servers to maximize the throughput.

Manage Large Number of Machines Efficiently: Data centers typically consist of a large number of machines serving multiple applications. As the number of machines to manage increases linearly, the dimensions of the state spaces increase at an exponential rate. Therefore, most dynamic resource provisioning frameworks face the problem of managing a large number of machines simultaneously. Our goal is to study the properties of the state search space as the number of machines is increased and minimize the growth of state search space.

The main difficulty in managing a large number of machines is the inability to efficiently predict a future state of the system. Predicting the behavior of a large scale distributed system often requires exploring millions of possible system state configuration. Furthermore, predicting the behavior of a system while looking multiple time steps into the future requires exploring tens of millions of states. Since exploring each state involves doing complex calculations such as realizing the revenue and cost associated with a specific state, making a single prediction may take a long time. Therefore, our goal is to be able to manage a large number of virtual servers while being able to predict can configure the behavior of a large cluster of servers.

Flexible Application Design: To study the behavior of resource provisioning frameworks under varying conditions, a resource provisioning

framework must be highly flexible. Therefore, in designing an application to simulate a dynamic resource provisioning framework, our goal is to make the application highly flexible and configurable. This application should be able to accept multiple user specified parameters such as number of threads, Horizon Levels, number of virtual machines and physical machines and simulate the framework.

Minimize SLA Violations: Dynamic resource provisioning frameworks make system resources available as they are needed. During time of low workload intensity, the right action for resource provisioning frameworks to take is to turn unused resources off. However, if the distributed system experiences a sharp increase in the incoming workload, it may not be able to respond quickly enough because of the time delay associated with turning on hosts and virtual machines. The inability to respond to system changes in time leads to Service Level Agreement (SLA) violations. A dynamic resource management system should be aware of the risk associated with turning off virtual and physical machines. Our goal is to develop a risk aware provisioning framework that can respond quickly to extreme changes in the workload and minimize SLA violations.

3.2 Object Design of Dynamic Resource Provisioning Framework

Dynamic Resource Provisioning Framework (DRPF) is an online resource provisioning framework that makes system resources available to client applications as needed. This framework maximizes system utilization by dynamically allocating virtual and physical machines to client applications. This dynamic allocation allows service providers to minimize power consumption and under-utilization of system resources. The DRPF also automatically tunes the system for optimal performance by computing an optimal frequency configuration

DRPF is a C++ application that uses objects oriented design principles to organize and manage its internal objects. This section provides a description of various DRPF objects, and how they are created and used.

LLC Object:

The Limited Lookahead Controller (LLC) class is the main driver in DRPF. This class instantiates virtual machines, physical machines and application clusters and manages them by keeping a track of their pointer. During the DRPF initialization phase, the LLC also generates the VM State Table, Frequency State Table, and PM State Table. The LLC class also generates and initializes the Prediction State Tree.

The two main functions of the LLC class are to measure the performance of the current system, and to predict an optimal system configuration. After a prediction is made, LLC proceeds by applying the prediction to the current system. The application process involves both turning on and turning off VMs and PMs. The VMs that are left on are then assigned appropriate frequencies for optimal performance. The LLC repeats this process after every time step until it finishes with a 24 hour period workload trace file.

Machine Object:

The machine class is the base object for a computer machine. Since both Physical machines and Virtual machines share common traits such as CPU, IP Address, and Computer Name, Machine object provides the basic functionality shared by both VMs and PMs. The Machine object includes data members such as Machine Name, State and CPU Configuration; and they are used to manage a machine's configuration. Figure 10 below shows the machine members.

Physical Machine Object:

The physical machine object represents a physical server and is derived from machine object. In addition to inheriting members from the base machine class, the physical machine class has additional members that primarily keep track of the virtual machines mounted on a physical host. The additional members include maximum number of VMs that the physical machine can handle, and a list of pointers to the virtual machines currently managed by the physical machine. Figure 10 below shows physical machine class's members.

Virtual Machine Class:

The virtual machine object represents a virtual server mounted on a physical machine. The virtual machine class is derived from machine class so it inherits the base functionality provided by the machine class. To associate a virtual machine with a physical host, the virtual machines store the physical machine's name and ID. The Virtual machine keeps track of the application they are serving by storing the application's name. Additionally, each virtual machine also keeps track of the processing rate and queue associated with the application. Figure 10 below shows virtual machine's class members.

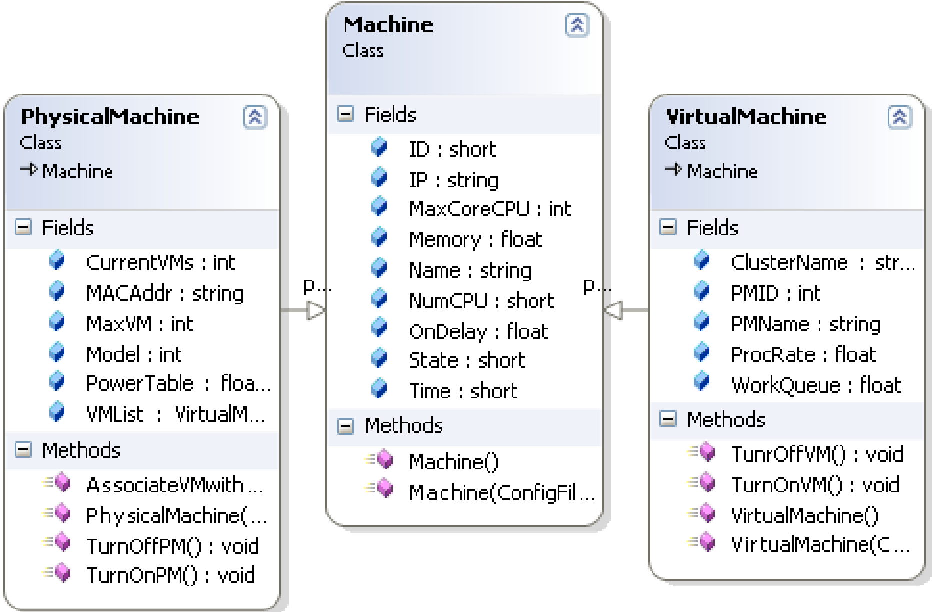

Figure 10: The Class Diagram of Machine, Physical Machine and Virtual Machine showing their members

Application Cluster Object:

The Application Cluster Object represents a client application. Each cluster object keeps track of the number of VMs and PMs currently associated with the client application. Since VMs and PMs are dynamically assigned to application clusters, they number of VMs and PMs assigned to each cluster changes over time. The Cluster object also holds the SLA related information such as dollars paid per request and the refund associated with SLA violations. Figure 11 below shows the class design of Cluster class.

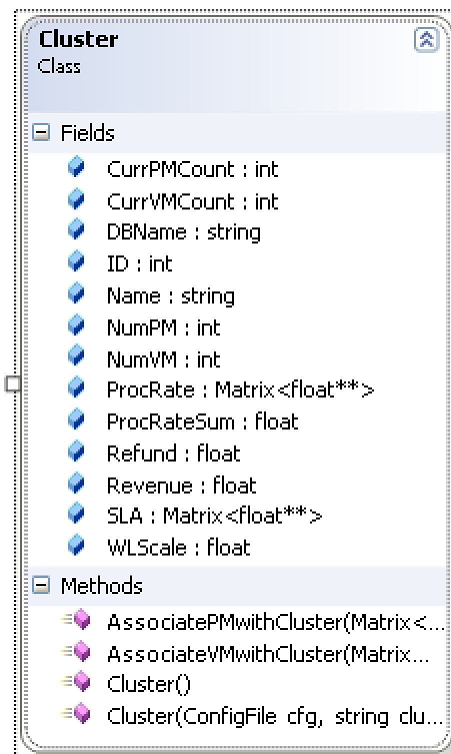

Figure 11: The Class Diagram of Cluster Class that represents client applications

VM State Table:

The VM State Table is a matrix of integers that contains all possible configurations of VM assignments to gold and silver applications. The table consists of a two dimensional array that is defined using the Matrix structure. Each row in the table consists of a unique VM configuration. The first column of a row contains the sum of all VMs that are assigned to gold application. The second row contains the sum of all VMs assigned to the silver application. The Middle columns consists of 1s, 2s and 0s where 1 means that the VM is serving Gold application, 2 means the VM is serving silver application, and 0 means that the VM is off and is not serving any application. The last few columns contains a sum of the VM that are ON and operational on a PM.

Figure 13 below shows an example of a small portion of VM Table that consists of 7 VMs and 2 PMs. This example shows how VMs can be partitioned among Gold and Silver application if we assign 6 VMs to Gold application and 1 VM to Silver application. The last two rows labeled PM1 and PM2 shows the sum of VMs that are currently mounted on Physical machine 1 and Physical Machine 2. In this specific example, since all VMs are ON, the sum of PMs mounted on PM1 and PM2 stays constant at 3 and 4 respectively.

		PM1			PM2					
Gold	Silver	VM1	VM2	VM3	VM4	VM5	VM6	VM7	PM1	PM2
6	1	1	1	1	1	1	1	2	3	4
6	1	1	1	1	1	1	2	1	3	4
6	1	1	1	1	1	2	1	1	3	4
6	1	1	1	1	2	1	1	1	3	4
6	1	1	1	2	1	1	1	1	3	4
6	1	1	2	1	1	1	1	1	3	4
6	1	2	1	1	1	1	1	1	3	4

Gold = 1
Silver = 2
Off = 0
Note: expert from VM State Table.

Figure 13: A small portion of a VM State Table that consists of all possible ways of assigning 6 VMs to Gold Application and 1 VM to Silver Application.

PM State Table:

The PM State Table is a matrix of integers that contains the current states of all PMs in our system model at any given time. Figure 14 below shows an example of PM State Table as different PMs are turned ON and OFF.

PM1	PM2	PM3	PM4	PM5	PM6	PM7
2	2	2	-2	-2	-2	-2
2	-1	-1	-2	-2	-2	-2
2	-2	-2	-2	-2	-2	-2
2	1	1	1	1	1	-2
2	2	2	2	2	2	-2

ON = 2
OFF = -2
BOOTING=1
SHUTTING DOWN = -1

Figure 14: The states of 7 PM over 5 time steps showing machines turning ON and OFF

Frequency State Table:

The frequency state table is a four dimensional matrix which consists of all possible configurations of frequencies for a given set of VMs and PMs. Figure 15 below shows a visual demonstration of the Frequency State Table. The w dimension in the table consists of PMs. The x dimension consists of the number of VMs that are currently ON and operational on the PM. The y dimension lists the different possible configurations for a given set of PM and VM combination. The z dimension consists of an array of individual frequencies for each VM.

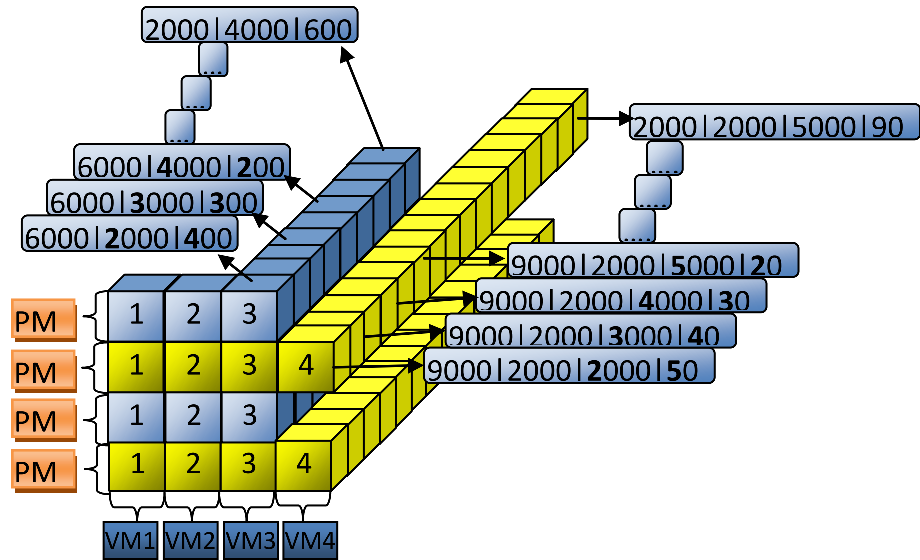

Figure 15: A 4-Dimensional Frequency State Table with four PMs

It's important to note that the Frequency State Matrix is not a square matrix and its w, x, y, and z dimensions are variable. For example, the x dimension for PM1 is three whereas it is four for PM4. Since the x dimension depends on the PM, we need a mechanism to keep track of the length of each dimension to prevent out of index exception. To achieve this goal, we have created an additional 2-dimensional table called the Frequency State Index Table. The index table keeps a track of the x and y direction. Figure 16 below shows a graphical illustration of the Frequency State Index Table.

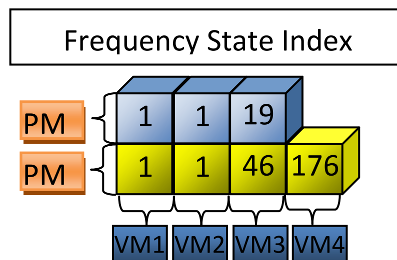

Figure 16: A 2-Dimensional Frequency State index table

State Search Tree Object:

The state object represents the state of our simulated distributed system at a given time. The state object takes snapshoot of the VM states and PM states at a given time. In our system model, the Root state always represents the current system state x(t) at horizon level 0. The state object constructs a 1-to-n tree where the pointers to the children are saved in a children matrix. The root state expands the state tree by creating n children states for x horizon level x(t+1); where n is equal to the total number of all possible VM configurations. For horizon level one and onwards, each state node creates just one child. Each state node keeps pointers to all of its children and single parent. Figure 17 below shows the State Tree Object graphical representation.

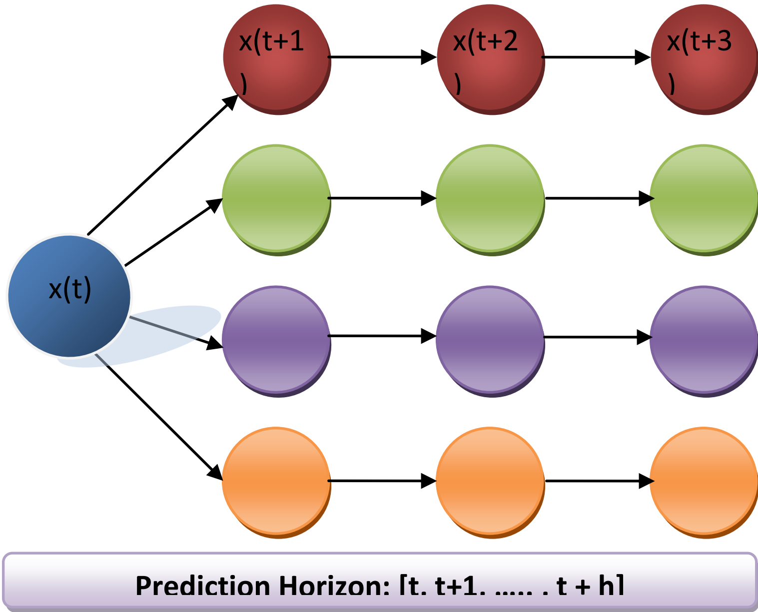

Figure 17: State Search Tree Object showing span of tree

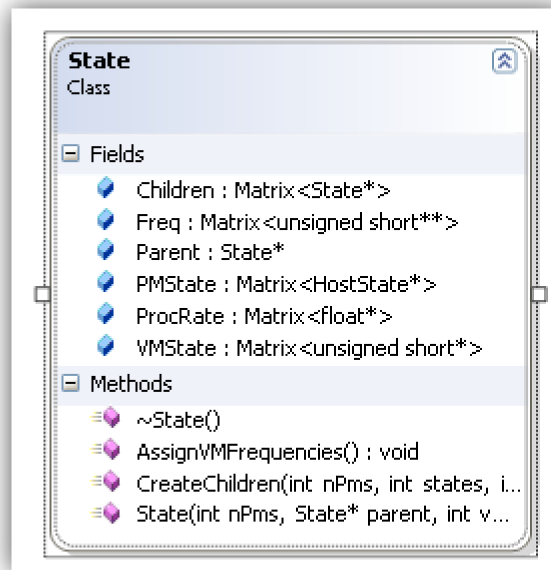

Figure 18: The Class diagram of State Object

System Configuration Class:

The DRPF application consists of hundreds of user defined parameters that are used to initialize the application. The configuration class allows us to read these parameters from a configuration file and load them into our application. This class allows us to read crucial user defined parameters such as number of threads, maximum horizon level, number of VMs and PMs. The configuration class makes DRPF application highly flexible and configurable which enables us to run the application using multiple configurations.

The Configuration File Reader class is an open source tool that has been developed by Rick Wagner from University of Michigan. The configuration class works by loading a .confg file into memory. To load a specific value from the configuration file, our application provides a search keyword. This file allows us to easily load hundreds of configurations from a configuration file. Figure 19 below shows the class design of Configuration File Class.

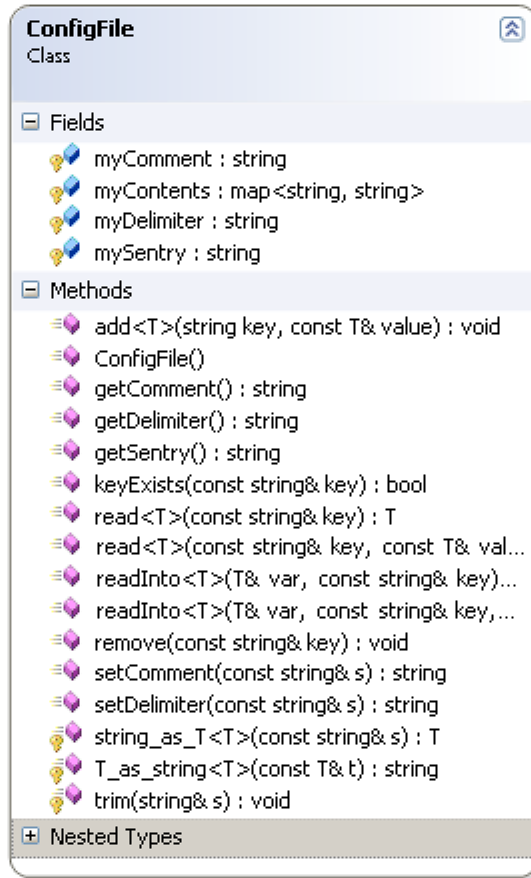

Figure 19: The class diagram of ConfigFile Class

3.3 DRPF Behavioral Design:

In the following section, we will discuss the behavioral design of DRPF.

Program Startup: The DRPF application begins by taking the file path of the configuration file and loading it into memory. The main function creates a new instance of the LLC class and sends the file path as a parameter. When the LLC is done with loading and initializing the system, the main function give the command "Go" to start the controller.

Initialization Machines and Hosts: To initialize the program data structures, the LLC default constructed calls the "Init" method. The init method read the configurations from the config file and loads them into

appropriate variables. Part of the initialization process includes instantiating virtual machines, physical machines, and client applications.

The LLC class reads the virtual machine configurations from the config file and instantiates virtual machine objects by calling virtual machine constructors. Since virtual machines are derived from machines objects, the machine class's default constructors are called also. During the instantiation phase, the virtual machines are assigned ids and names.

The LLC class reads the physical machine configurations from the config file and instantiates physical machine objects by calling physical machine constructors. Since physical machines are derived from machines objects, the machine class's default constructors area also called. During the instantiation phase, the physical machines are assigned ids and names. As an additional step in the initialization process, the physical machines search through the list of virtual machines and insert a pointer of the VM it is hosting into its VMList array.

The LLC class reads the client application cluster configurations and creates two client applications. After loading cluster information, the cluster continues by searching through the VM and PM lists and inserts a pointer of the VMs and PMs It is currently using to process the workload into its VM and PM lists.

Creating Frequency State Table: The frequency configuration state table consists of all possible set of frequencies that can be assigned to virtual machines mounted on a given physical host. To limit the size of frequency state table, we impose implicit constraints on the table. We use the following constraints to develop the Frequency State Table.

1. The minimum possible frequency on a VM is 2000 MHz.
2. The maximum possible frequency on a VM cannot exceed more than half of the maximum frequency on the physical host the VM is mounted on.
3. The minimum frequency step is 1000 MHz.

4. The sum of the frequencies assigned to VMs on a single host must be exactly equal to the maximum frequency of the physical machine.

Creating VM State Table: The VM state configuration table consists of all possible ways of assigning the virtual machines to two client application. To divide the VMs among two applications, we first consider the total number of VMs we can assign to two applications. The total number of VM assigned to each application is limited by the following constraints.

1. At any given time, each application must have at least one operational VM.
2. The sum of the VMs assigned to each application cannot exceed the total number of VMs in the system.

Using these constraints, we can develop a table of valid VM assignments to two applications. Figure 20 shows the number of VMs assigned to each application given a maximum of 7 VMs.

The table in figure 21 shows a set of possible ways to assign the total number of VMs to gold and silver applications. There are a number of ways to map each set (e.g. {1,1} where {gold, silver}) to VMs. For example, given five VMs and a {4,1} assignment set, there are 5 different ways of assigning the VMs to two applications. Figure x below show a small example of how to assign applications to different VMs.

Gold = 1, Silver = 2					
1 2	1 2	1 2	1 2	1 2	1 2
1 1	2 1	3 1	4 1	5 1	6 1
1 2	2 2	3 2	4 2	5 2	6 2
1 3	2 3	3 3	4 3	5 3	6 3
1 4	2 4	3 4	4 4	5 4	6 4
1 5	2 5	3 5	4 5	5 5	6 4
1 6	2 6	3 6	4 6	5 6	6 5

Legend:
- Valid State
- Valid Inverted State
- Invalid State

Max VM = 7

Figure 20: VM State Table application VM assignments

		Gold = 1, Silver = 2, Off = 0				
Gold	Silver	vm1	vm2	vm3	vm4	vm5
4	1	1	1	1	1	2
4	1	1	1	1	2	1
4	1	1	1	2	1	1
4	1	1	2	1	1	1
4	1	2	1	1	1	1
3	1	1	1	1	2	0
3	1	1	1	2	1	0
3	1	1	2	1	1	0
3	1	2	1	1	1	0
3	1	1	1	1	0	2
3	1	1	1	2	0	1
3	1	1	2	1	0	1
3	1	2	1	1	0	1
3	1	1	1	0	1	2
3	1	1	1	0	2	1
3	1	1	2	0	1	1
3	1	2	1	0	1	1
3	1	1	0	1	1	2
3	1	1	0	1	2	1
...	...	...	...	...	...	...
...	...	...	...	...	...	...
...	...	...	...	...	...	...
...	...	...	...	...	...	...
...	...	...	...	...	...	...

Figure 21: An example of VM State Table that consists of all possible ways of assigning VMs to client applications

Creating State Search Tree: The State Tree is used in the LLC prediction algorithm. The initialization procedure starts from the root state and creates n children where n is equal to the number of VM states. The controller continues by creating a single child for each of the children states until the maximum horizon level is reached. Each child node is assigned a VM state. Figure 22 below shows the algorithm used to create the State Search Tree. A visual representation of the state search tree is shown in figure 17.

```
printf("Initializing State Table \n");
RootState = new State(Pms.x, NULL, VMTbl.y);
RootState->CreateChildren(Pms.x, VMTbl.x, VMTbl.y);

if (MAX_HORIZON > 1)
{
    for (int h1 = 0; h1 < RootState->Children.x; h1++)
    {
        State *c1 = &RootState->Children.node[h1];
        c1->CreateChildren(Pms.x, 1, VMTbl.y);

        State *c2 = &c1->Children.node[0];
        c2->CreateChildren(Pms.x, 1, VMTbl.y);

        State *c3 = &c2->Children.node[0];

        c1->VMState.node = VMTbl.node[h1];
        c2->VMState.node = VMTbl.node[h1];
        c3->VMState.node = VMTbl.node[h1];
    }
}
```

Figure 22: The Algorithm used to create the State Search Tree

Loading Incoming and Predicted Workload Files: The workload files contain the incoming user requests in 120 second increments. Our program loads these comma-separated files into memory and multiplies the workload with a scalar to adjust the workload intensity. Our system model is designed to adequately handle the maximum workload intensity for both applications simultaneously. For example, the maximum predicted workload is 70,602 and 13,408 requests per 120 seconds for both gold and silver applications, respectively. The gold application needs at least 24 VMs to process its maximum workload and the silver application needs at least 19 VMs to process its workload. The sum of the minimum number of VMs need to process the workload is 43 VMs; and our model consists of 44 VMs. Therefore, at any given time, we should always have enough VMs to process the workload.

Until this point, our application is involved in initializing various system parameters. Beginning from the following step, the main controller begins to simulate the workload for a 24 hour period. The controller will repeat

the steps listed below until it is done with simulating the workload. The controller comes up after every 120 seconds and tunes our modeled distributed system for optimal performance and minimal power consumption. The controller is designed to predict the future state and apply the state to the system within 10 seconds.

Updating Physical Machine States: The controller first updates the physical machine states. If a PM is given the order to shut down, the controller checks to see if all of the VMs mounted on the PM have been turned off successfully. If all VMs are off, the controller changes the state of the system to "ready to shut down" but does not give the order to shut down the PM immediately. It uses passive control and allows the prediction algorithm to first predict the next state of the system. If the next state requires the PM to be on and operational, then the controller can easily turn on the VMs. However, if the next state does not use the current PM, then this PM is given the order to begin shutting down.

Delaying the order to shut down the PMs allows the controller to better handle turbulent workload and minimize potential SLA violations. When it comes to turning on a Physical Machine, the controller gives the order to turn on a PM as early as possible. If it is needed, the main objective of the controller is to have a machine up and operational as soon as possible.

Updating Virtual Machine States: After turning on or off PMs, the controller moves on to turn ON or OFF VMs. If the previous prediction suggested that a specific VM is not needed, and one time step has passed since the last prediction, the controller will update the state of the VM as OFF. Turning a VM off saves energy and helps lower down cost.

Obtaining System Dynamics: Once our system model's PM and VM states have been updated, the controller moves on towards handling the incoming workload. The controller first sums up the total processing capabilities of each application by iterating over all of the VMs. Note that the controller skips the VMs that have been turned off. For any VM that is on, the controller computes its processing rate based on its current operating frequency and current queue. Then, depending on the

application the VM is serving, the processing rate is added to the total processing power for that application. Calculating the processing rate sum for each application helps us in distributing the workload among active VMs.

Using the processing rate sum for client applications, the controller computes the workload share for each VM using the following function.

$$Work\ Load\ Share\ for\ VM_i = \frac{\text{Proc Rate of VM}_\text{i}}{\text{Proc Rate Sum of App VM}_\text{i}\text{ is Serving}}$$

The function listed above allows us to distribute the incoming workload evenly among processing VMs. The controller next computes the next queue for the VM using the following function.

$$Q(t+1) = Q(t) + \left(\frac{WL(t) \times WLfrac}{120\ s} - \frac{Proc\ Rate\ of\ VM_i\ \times (120\text{s})}{s}\right)$$

where Q represents the queue and WL represents the fraction of the workload assigned to VM_i . The controller next computes the revenue generated by the specific VM by multiplying the number of requests processed by the SLA Revenue function. If there is a queue, a refund is given to the client by subtracting from the revenue.

The power consumed by the distributed system depends on the number of PMs and the number of VMs that are currently on. The controller iterates over each PM and computes its power consumption that also takes into account the number of VMs on. The figure below shows the code used in estimating the system cost.

```
//Calculate cost by going over all of the PMs.
for (int pm = 0; pm < Pms.x; pm++)
{
    float pmPower = GetPowerConsumption(Pms.node[pm], Pms.node[pm]->CurrentVMs);
    Cost = Cost + EvalCost(pmPower);
}
```

Figure 23: The Algorithm used to compute cost

3.4 Controller Design and Implementation

After handling the current workload, the controller moves on towards predicting the future state of our system. The RootState denotes the current system state. The controller calls the function SingleStateExpansionController() with references to current queue and current system state. This function recursively calls itself until the maximum horizon level is reached. It effectively does the job of nested FOR loops without using nested statements. The recursive design allows us to easily control the depth of horizon levels without the need to make any code changes. Figure below shows a functional design of this function (Note that this is not the complete code, the actual function is 700 lines long).

```
void LLC::SingleStateExpansionController(int hIndex, Matrix<float*> queue,
    float earnings, State *parentState)
{
    if (hIndex == 1)
    {
        #pragma omp parallel default(none) private(currPM, pmNvms).....
        {
            #pragma omp for ....
            for (state = 0; state < parentState->Children.x; state++)
            {
                <Explore All States at horizon level 1>;

                SingleStateExpansionController(hIndex+1,queue, earnings, currentState)
            }
        }
    }
    else
    {
        <explore all states at horizon level 2 - Max Horiozn>;

        if (hIndex < MAX_HORIZON)
            SingleStateExpansionController(hIndex+1, queue, earnings, currentState)

        if (hIndex == MAX_HORIZON)
        {
            #pragma omp critical
            {
                if (currentBranchEarnings > HighestErng.Earnings)
                {
                    HighestErng.Earnings = currentBranchEarnings;
                    HighestErng.H1State = currentBranchH1State;
                }
            }
        }
    }
}
```

Figure 24: An outline of the parallelized prediction algorithm

The prediction algorithm begins by looking at the predicted workload and computing the minimum number of VMs needed to process the incoming workload for each application. To compute the minimum number of VMs needed for each application, the controller takes the Average VM frequency, which in our cluster is 4000 MHz, and computes the processing rate for each application.

Next, the controller adds the predicted workload with the queue to compute the total requests each application has to process. The controller then divides the total requests that each application must process by the average processing rate for each application to come up with the minimum number of VMs needed to process the workload. The code listed in figure 25 shows how we compute the minimum number of VMs needed for gold and silver applications.

To improve the performance of our controller; we parallelize the state exploration process. We use OpenMP to create and manage threads. At horizon index 1, we assign a thread to a state and explore it for three horizon levels. When the thread is done with exploring the current state over three horizon levels, it moves on and obtains a new state to explore. Since the controller must explore millions of VM states at each time step, parallelizing the state exploration process significantly improves the performance of the controller.

Note that at horizon level 2 and 3, the controller explores the same state that was selected in horizon level 1. The controller explores horizon levels 2 and 3 with the same thread also. So for example, Thread 1 will explore VM State 1 at all horizon levels. When thread 1 gets done with exploring state 1, it moves on to select the next available state that is not being currently explored by another thread. In the figure 26 shown below, thread 1 sees that VM State 3 is not being explored so it select VM3 and explores it over all three horizon levels.

```
float goldProcRate = GetProcessingRate(AvgFreq, queue.node[0], AppCluster.node[0]);
float silverProcRate = GetProcessingRate(AvgFreq, queue.node[1], AppCluster.node[1]);

MinVMReq[hIndex-1][0] = (WL.node[0][hIndex]+queue.node[0]*1.2) / (goldProcRate * STEP);
MinVMReq[hIndex-1][1] = (WL.node[1][hIndex]+queue.node[1]*1.2) / (silverProcRate * STEP);
MinAppProcRate[hIndex-1][0] = MinVMReq[hIndex-1][0] * goldProcRate;
MinAppProcRate[hIndex-1][1] = MinVMReq[hIndex-1][1] * silverProcRate;

float minGoldVMs = ceil(MinVMReq[hIndex-1][0]);
float minSilverVMs = ceil(MinVMReq[hIndex-1][1]);
```

Figure 25: The Algorithm used to compute the minimum number of VMs needed for gold and silver applications

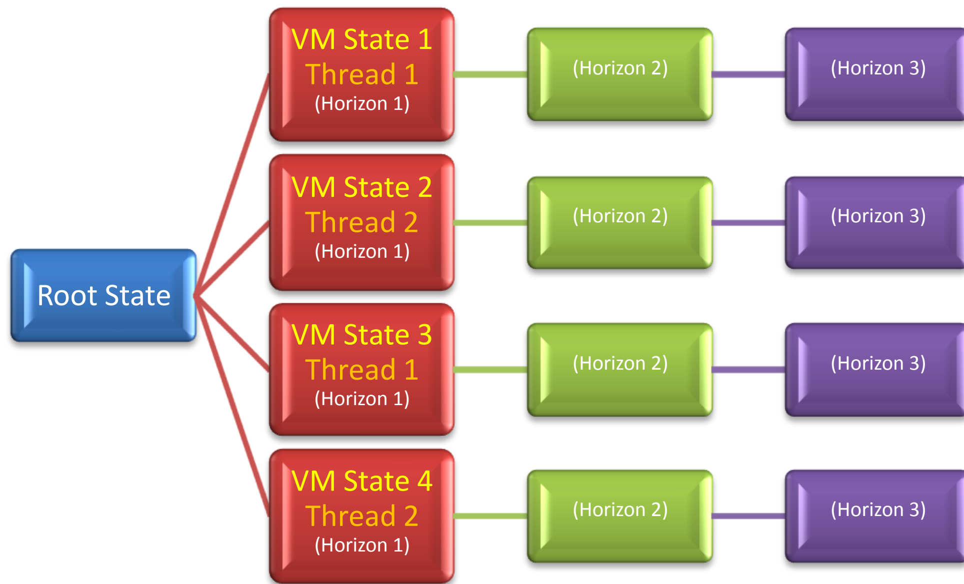

Figure 26: An example showing how multiple threads explore different VMs states in parallel

The VM state search space is huge and exploring the entire search space (even when parallelized) will take a long time. To reduce the number of states that are fully explored, the controller limits the number of states explored at horizon level 1 by eliminating unneeded states. for example, if the controller predicts that (described in step I) it needs 15 VMs for gold application and 12 VMs for silver application, then it makes no sense to explore a state that allocated 10 and 8 VMs to gold and silver applications, respectively. Our controller explores only the states that have at least enough VMs to process the workload. Figure 27 below shows the code segment that limits the number of VM states explored at horizon level 1.

```
bool enoughVM = true;
if ((gold_CurrStateVMs < (minGoldVMs)) ||
    (gold_CurrStateVMs > (minGoldVMs + Delta)) )
    enoughVM = false;
else if ((silver_CurrStateVMs < (minSilverVMs)) ||
    (silver_CurrStateVMs > (minSilverVMs + Delta)) )
    enoughVM = false;
```

Figure 27: The Algorithm used to limit the number of states explored for prediction purposes

For a given VM state combination, the controller explores the Frequency state search space table to find an optimal combination of VM frequencies that will maximize the throughput. To find the optimal frequency combination, the controller assigns the frequencies to VMs and observes their processing rate. If all of the VMs can process the predicted workload without generating any queues, then an optimal frequency configuration has been obtained.

For a given VM state combination and a Frequency state combination, the controller computes the cost and revenue associated with the VM state. The controller computes the cost and revenue at each horizon level and passes these values to the next horizon level. For example, if the revenue generated at horizon level 1 is $2 and cost is 2 cents, it will pass this information to horizon level 1. If the revenue and cost at horizon level is $3 and 3 cents, then the controller will pass $5 and 5 cents as the cost to horizon level 3.

As the controller threads moves from one horizon level to the next, they sums up the revenue and cost associated with a specific VM states. After a thread is done with exploring the last horizon level, the thread compares its earnings (revenue – cost) with the highest earnings obtained by all threads so far. If the thread A has higher earnings, then it updates the highest earnings state to point towards the state that thread A is exploring. This process continues until all states have been explored.

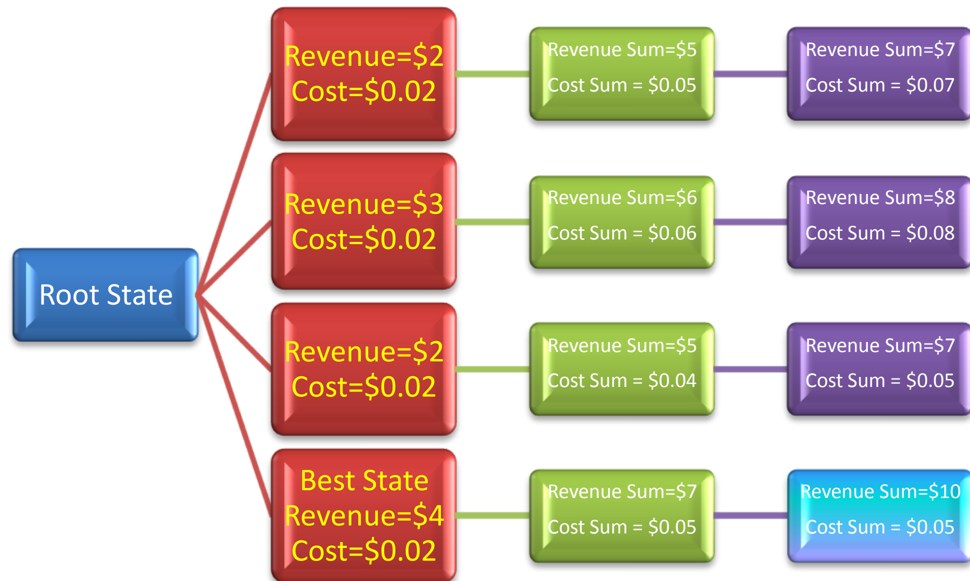

Figure 28: An example showing how the best state is chosen among many alternatives

When the controller is done with exploring all states, the highest earning state will point towards the state with highest earnings. The highest earnings state is the best predicted state of the system.

After the controller has successfully predicted the state with highest earnings, the controller next applies the predict state to the current system. The application process involves the following steps.

1. Based on the predicted VM states, give the signal to turn VMs ON or OFF.
2. Based on the predicted PM states, give the signal to turn PMs ON or OFF.
3. Apply the optimal frequencies to each VM that in the cluster.

It's important to note that before a PM can be turned off, all of its VMs must be turned off.

The controller comes up after every 120 seconds and repeats the process listed in steps 7 through 13. The controller repeats this process for a period of 24 hours.

Chapter 4
Simulation Results

In this chapter we explore the results obtained by simulating our dynamic resource provisioning framework. We will first discuss the power savings associated with dynamically turning ON and OFF virtual machines and physical machines. Next, we will discuss the benefits of parallelizing the controller's search routine to enhance the performance of our DRPF.

4.1 Control Performance

The results from our DRPF simulations show that dynamically allocating system resources can reduce power consumption by as much as 30% over a 24 hour period. During periods of low activity, the power savings can grow as high as 59%. The power savings are realized by turning off virtual machines and physical machines in response to lower workload intensity.

In an uncontrolled system, the resource providers leave PMs and VMs ON. Therefore, the power consumption of uncontrolled systems stays constant over time. On the contrary, our system turns unneeded resources off to conserve power. Figure 29 below shows the power consumption of both controlled and uncontrolled systems.

Comparing the power consumption of controlled system with the workload intensity shows a very close correlation between the workload intensity and the amount of power used by the system. As the workload decreases, the power consumption decreases; and as the workload increases, the power consumption goes to 100%.

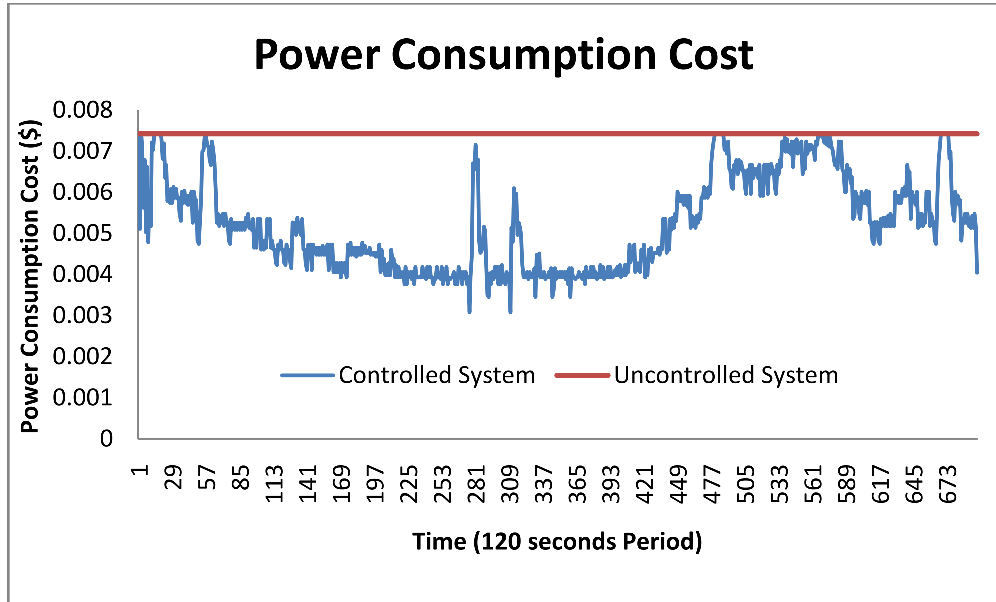

Figure 29: Power Consumption comparison of controlled and uncontrolled systems

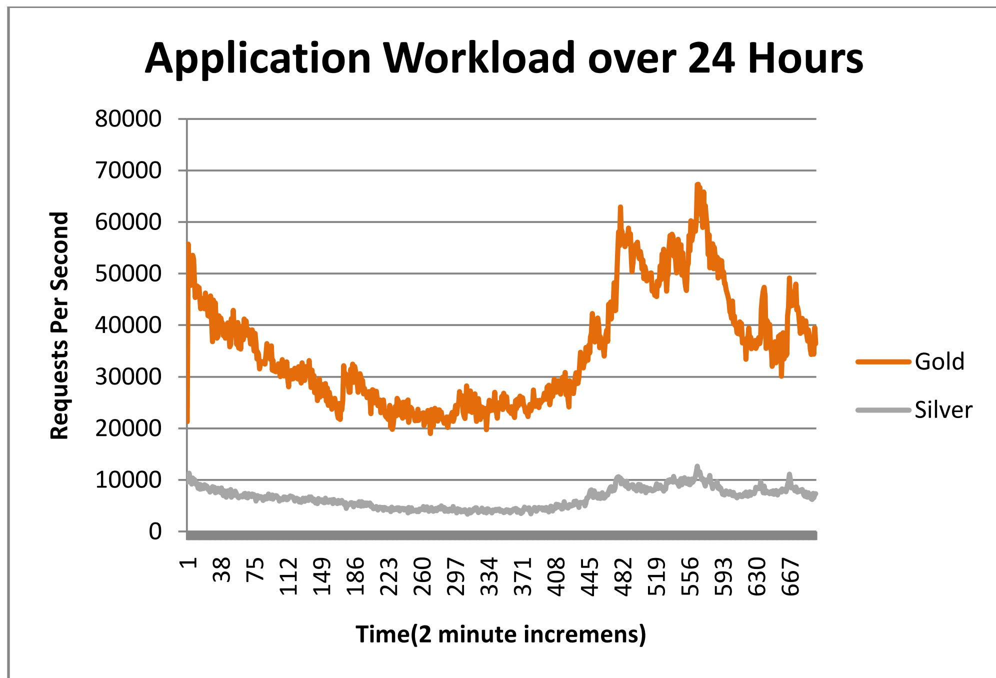

Figure 30: Incoming Workload for both gold and silver application

Over a 24 hour period, the graph of number of PMs operating very closely correlates the workload intensity graph. Figure 31 below shows that as the number workload intensity decreases, the number of PMs ON decreases also.

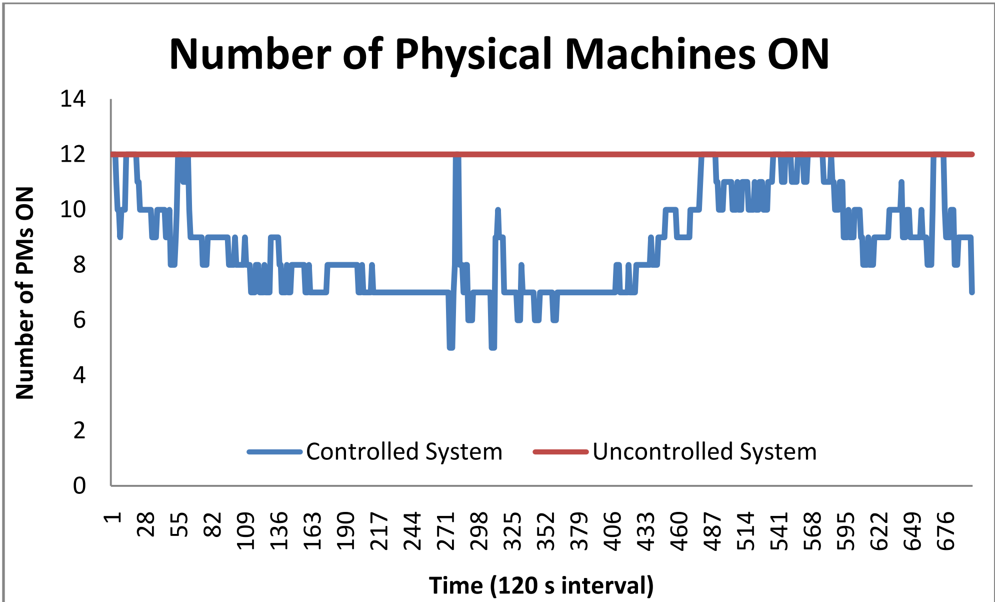

Figure 31: Number of Physical Machines ON and OFF during 24 hours period

The virtual machine utilization graph shows that the number of VM utilized in the system fluctuates in response to workload fluctuations. Figure 32 below shows the number of VMs operating as the incoming workload increases and decreases.

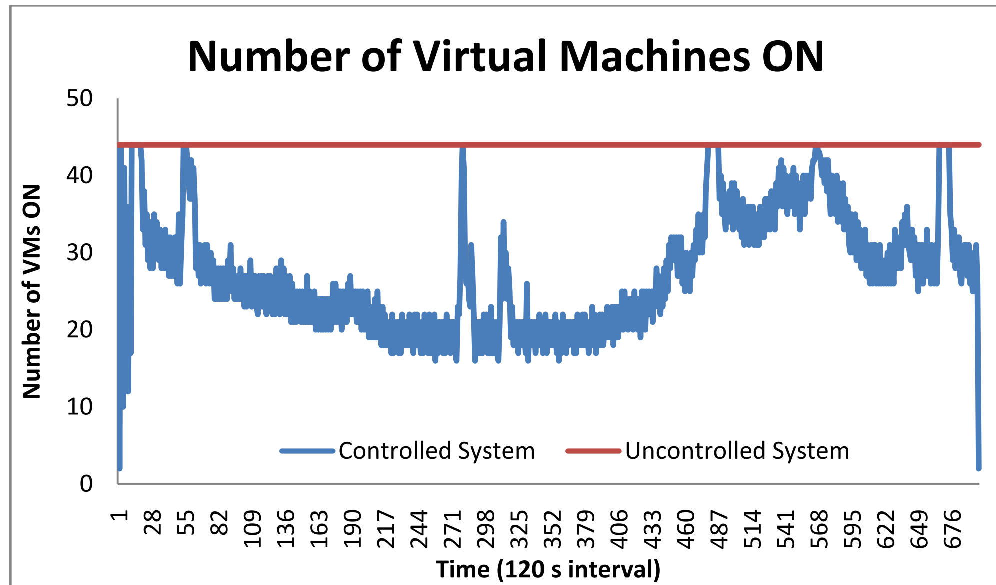

Figure 32: Number of Virtual Machines ON and OFF during 24 Hours Period

4.2 Execution Speedup via Parallelization:

Our results show that parallelizing the LLC's state exploration routine improves the performance by as much as 263%. Parallelizing allows us to make a control decision from millions of states within just 4.5 seconds. This performance improvement is obtained by running the DRPF application on a virtualized quad-core machine. Figure 33 below shows a decrease in average prediction time as the number of threads is increased.

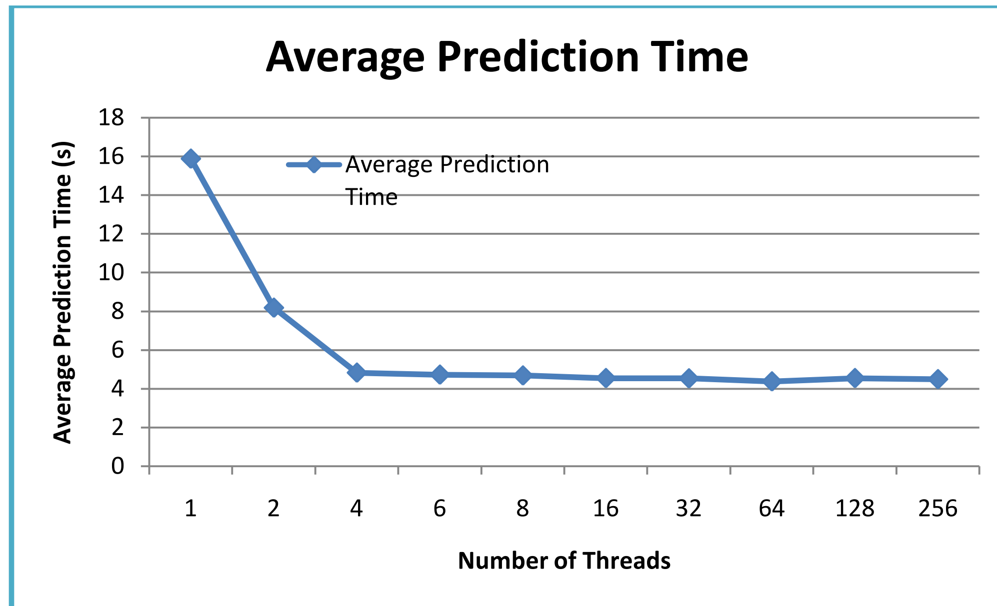

Figure 33: The Average Prediction Time of the controller VS. number of threads

It's interesting to note that we see a significant improvement in the average prediction time as the number of threads is increased from 1 to 2 to 4. However, after 4, we see minimal performance improvement. This is due to the fact that our simulation environment consists of a virtual machine with 4 cores. Our controller can run 4 threads in parallel but it cannot run more than 4 threads in parallel. Therefore, as the number of threads is increased, the additional threads have to wait before they can be executed.

It's important to note that as the number of threads is increased from 4, we still see a minute performance improvement. This is due to the fact that at 4 threads, the processer utilization goes to an average high of 90%, but it rarely reaches 100% utilization. As the number of threads is increased from 4, the processor utilization begins to go up to 100%.

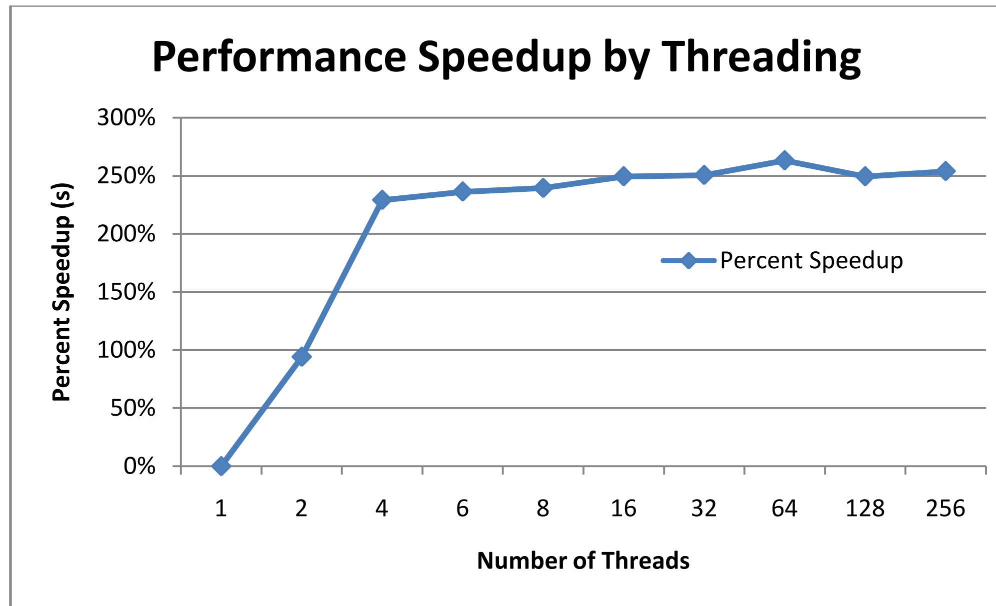

Figure 34: Execution Speedup obtained via parallelization of the prediction algorithm

Chapter 5
Conclusions

Being able to dynamically allocate the resources of large scale distributed systems based on the arriving workload can save millions of dollars for many large corporations such as Google and IBM. We have shown a practical approach to the dynamic resource provisioning problem that can reduce server underutilization and minimize the power cost. A parallelized Limited Lookahead Controller provides these capabilities by efficiently using the available system resources for optimal performance. The prediction algorithm of the LLC is a computation intensive process it can take a long time to predict the optimal state for a large system. As the size of the distributed system increases, the possible state configurations begins to increase exponentially and the response time of the sequential LLC begins to decrease. Parallelizing the LLC with OpenMP shows significant performance enhancement. The performance gain is further enhanced by the use of code optimization strategies. With the use of more powerful hardware, the performance of the LLC increases even further. The strategies detailed in this paper proves that the use of parallelization techniques substantially improve the performance of the LLC.

Bibliography

[1] S. Abdelwahed, N. Kandasamy, and S. Neema. *A Control-Based Framework for Self-Managing Distributed Computing Systems*. IEEE Real-Time & Embedded Tech.& Applications Symp.,2004.

[2] D. Kusic, J. O. Kephart, J. E. Hanson, N. Kandasamy, G. Jiang. *Power and Performance Management of Virtualized Computing Environments Via Lookahead Control*. ICAC, 2008.

[3] *A. C. Harvey. Forecasting Structural Time Series Models and the Kalman Filter. Cambridge University Press, Cambridge, 1989.*

[4] *D. Kusic, N. Kandasamy, S. Abdelwahed . Risk-Aware dynamic Resource Provisioning in Enterprise Computing Systems. IEEE Conference on Autonomic Computing, June 2006.*