SCALABLE AND ROBUST COMPUTE CAPACITY MULTIPLEXING IN VIRTUALIZED DATACENTERS

A Thesis
Presented to
The Academic Faculty

by

Mukil Kesavan

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology
August 2014

SCALABLE AND ROBUST COMPUTE CAPACITY MULTIPLEXING IN VIRTUALIZED DATACENTERS

Approved by:

Professor Karsten Schwan,
Committee Chair
School of Computer Science
Georgia Institute of Technology

Professor Karsten Schwan, Advisor
School of Computer Science
Georgia Institute of Technology

Dr. Ada Gavrilovska
School of Computer Science
Georgia Institute of Technology

Professor Calton Pu
School of Computer Science
Georgia Institute of Technology

Professor Douglas M. Blough
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Ravi Soundararajan
Principal Engineer
VMware Inc.

Date Approved: 5 May 2014

To my mother.

ACKNOWLEDGEMENTS

There are a lot of people that have helped me, both personally and professionally, during my years in graduate school at Georgia Tech. First and foremost, my advisors, Prof. Karsten Schwan and Dr. Ada Gavrilovska, have been a source of never ending support over the years. They've provided me with invaluable mentorship on turning abstract ideas into well designed systems, that stand the test of a rigorous evaluation, and on elegantly sharing those ideas with the outside world as research publications. I owe a lot to the level of patience and faith they showed in me during the ups and downs of the journey towards this dissertation.

I thank the remaining members of my dissertation committee, Prof. Calton Pu, Prof. Douglas Blough and Dr. Ravi Soundararajan, for their insightful comments on my research and ideas for future work. My mentors during my internships at VMware Inc., Orran Krieger, Irfan Ahmad and Ajay Gulati, have had a huge impact on the ideas presented in this thesis. I also thank VMware Inc., for generously donating software licenses and technical know-how that allowed me to evaluate my systems at very large scales. Chad Huneycutt, our awesome systems administrator, also deserves much approbation for all the time he spent with me making sure I have the massive infrastructure ready and running smoothly for my experiments.

I've had the pleasure of several intellectually stimulating discussions with my lab mates over the years: Adit Ranadive, Priyanka Tembey, Vishakha Gupta, Qingyang Wang, Dulloor Rao, Hrishikesh Amur, Vishal Gupta, Min Lee, Chengwei Wang, Fang Zheng, Liting Hu, Minsung Jang, Jay Lofstead, Hobin Yoon, Alex Merritt, Sudarsun Kannan, Ketan Bhardwaj, Dipanjan Sengupta and Junwei Li. In addition to being research colleagues, several of them have been and still are also dear friends. My

interactions with them made sure that I was also on top of the latest research in all related areas of systems.

There are numerous other friends outside of my immediate academic circle that are responsible for my having a truly enjoyable experience living in Atlanta. I'm thinking of every one of you as I write this and I thank you very much for being a part of my life through graduate school. Finally, I thank my family for their understanding, love and affection through every crazy adventure I've pursued thus far, including a doctoral degree.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Multi-tenant cloud computing datacenters run diverse workloads, inside virtual machines (VMs), with time varying resource demands. Compute capacity multiplexing systems dynamically manage the placement of VMs on physical machines to ensure that their resource demands are always met while simultaneously optimizing on the total datacenter compute capacity being used. In essence, they give the cloud its fundamental property of being able to dynamically expand and contract resources required on-demand.

At large scale datacenters though there are two practical realities that designers of compute capacity multiplexing systems need to deal with: (a) maintaining low operational overhead given variable cost of performing management operations necessary to allocate and multiplex resources, and (b) the prevalence of a large number and wide variety of faults in hardware, software and due to human error, that impair multiplexing efficiency. In this thesis we propound the notion that explicitly designing the methods and abstractions used in capacity multiplexing systems for this reality is critical to better achieve administrator and customer goals at large scales.

To this end the thesis makes the following contributions: (i) CCM - a hierarchically organized compute capacity multiplexer that demonstrates that simple designs can be highly effective at multiplexing capacity with low overheads at large scales compared to complex alternatives, (ii) Xerxes - a distributed load generation framework for flexibly and reliably benchmarking compute capacity allocation and multiplexing systems, (iii) A speculative virtualized infrastructure management stack that dynamically replicates management operations on virtualized entities, and a compute

capacity multiplexer for this environment, that together provide fault-scalable management performance for a broad class of commonly occurring faults in large scale datacenters.

Our systems have been implemented in an industry-strength cloud infrastructure built on top of the VMware vSphere virtualization platform [27] and the popular open source OpenStack [17] cloud computing platform running ESXi and Xen [37] hypervisors, respectively. Our experiments have been conducted in a 700 server datacenter using the Xerxes benchmark replaying trace data from production clusters, simulating parameterized scenarios like flash crowds, and also using a suite of representative cloud applications. Results from these scenarios demonstrate the effectiveness of our design techniques in real-life large scale environments.

CHAPTER I

INTRODUCTION

Cloud computing has become a popular computing paradigm that allows end-users to dynamically scale up or down the resources they use to run their applications. Virtualization of resources is a key enabler of such fluid mapping of infrastructure compute capacity. Typically, users pay only for resources they actually use, resulting in large cost savings compared to self-hosting applications on dedicated hardware. Cloud providers build large scale datacenters to exploit cost advantages due to economies of scale and strive to oversubscribe available compute capacity and statistically multiplex it between applications to maximize operational efficiency and pass on the cost benefits to consumers.

However, the reality at several public and private cloud datacenters is that they seldom, if ever, employ capacity multiplexing on an ongoing basis. Several factors such as increased monitoring and actuation overheads at scale, prevalence of failures, hardware heterogeneity, hard and soft virtual machine (VM) placement constraints, resource partitions etc. pose challenges to their use.

Popular public cloud services like Amazon EC2 [2], and Rackspace Cloud [19], simply do not oversubscribe resources and dynamically multiplex the compute capacity between applications [88, 20]. They typically allow customers to scale up or scale down their resources in coarse-grained units of individual VMs while statically reserving compute capacity allocated to each VM. This leads to two problems: (i) despite virtualization and consolidation of VMs, cloud datacenters still do not achieve good server utilization levels [58], (ii) the coarse-grained scaling support works only for embarrassingly parallel and stateless applications, leaving out the huge market of

enterprise applications like CRM, ERM, SCM etc. that do not scale horizontally [69].

Even in private enterprise datacenters, commercially available infrastructure resource allocation and multiplexing software such as Microsoft PRO, VMware DRS, and Citrix XenServer currently only support dynamic allocation for a set of 32 or fewer hosts [14, 6, 30]. In addition, to foregoing opportunities to improve datacenter server utilization levels, capacity multiplexing within such small scales is also ineffective in supporting the resource needs of applications with highly fluctuating demands that may require large amounts of capacity during flash crowd scenarios.

Given the scale and complexity of the deployed software at various layers, and the ubiquity of commodity hardware in modern cloud environments, failures of the operations needed to multiplex and allocate resources (e.g. VM reconfiguration and migration) are far too common and they serve to severely limit the gains achievable through capacity multiplexing. Notwithstanding fail-stop hardware faults, several prior studies have pointed out a wide variety of causes underlying service failure in hosting centers such as software bugs, misconfiguration, state inconsistencies, limpware, resource insufficiency etc [38, 84, 66, 54, 67, 78].

These failures contribute to a 'glass ceiling' in the datacenter management plane that limits the improvements achievable by capacity multiplexing solutions [70, 81]. In addition, the cost of carrying out the management actions necessary to allocate resources, such as migrating a virtual machine (VM), is often much greater than the algorithmic cost of computing actions [94]. This limitation can directly influence the convergence, and hence, the design of multiplexing algorithms [68, 63].

In this thesis, we address the problem of multiplexing the compute capacity of large scale datacenters as a whole. Specifically, we identify and scope our work to tackle two key challenges: (i) scalability - maintaining low overhead for monitoring and actuation of operations needed to allocate capacity, (ii) fault-tolerance - ensuring fault-scalable performance in the presence of a wide variety of failures. We approach this task by

building, deploying and evaluating systems at large scales. Then, by analyzing the large amounts of data collected during the process to iteratively refine the design of our systems. Such an approach also required the development of benchmarking tools that permit the proper exploration of the design space - a role that none of the existing benchmarking tools perform satisfactorily.

We believe that practical compute capacity multiplexing solutions can employ simple design methods that are easy to develop, debug and scale in lieu of more complex approaches owing to the inherent limits discussed above. They should also strive to adapt their design to treat failure of management operations as normal operating conditions and achieve good resource allocation and multiplexing performance despite their prevalence.

1.1 Thesis Statement

Compute capacity multiplexing solutions can achieve effective and fault-scalable multiplexing at large scales by making dynamic tradeoffs between allocation accuracy and management enforcement overhead and by speculatively replicating management operations across different physical targets.

1.2 Contributions

To validate the thesis, we make the following contributions:

(i) **CCM (Cloud Capacity Manager)** is an on-demand compute capacity multiplexing system that combines various low-overhead techniques, motivated by practical on-field observations, to achieve scalable multiplexing across thousands of machines. CCM achieves this scale by employing a 3-level hierarchical management architecture. The capacity managers at each level continuously monitor and aggregate *black-box* VM CPU and memory usage information and then use this aggregated data to make independent and localized capacity allocation decisions. The core concept embodied in CCM's capacity multiplexing is the on-demand balancing of load, across logical

host-groups, at each level in the hierarchy. Reductions in management cost are obtained by having monitoring and resource changes occur at progressively less frequent intervals when moving up the hierarchy, i.e., at lower vs. higher level managers. As mentioned previously, hardware and software faults are extremely common at large scales, and the cost of reconfiguration, i.e., performing VM migrations, is fairly expensive [94] and non-deterministic [93]. These factors limit the efficiency of accurate and high overhead multiplexing methods. CCM takes this into account in its design and, hence, uses a simple greedy hill-climbing algorithm to iteratively reduce load imbalances across the hierarchy in lieu of more complex approaches. It also uses management operation throttling and liberal timeouts, to select cheaper management actions, and, minimize the incidence of management failures due to network resource overuse. Such simple methods are easier to scale, develop and debug, and, their effectiveness is shown through detailed large scale experiments on a 700 node cluster running the VMware vSphere virtualization platform.

(ii) A detailed study and analysis of the cost and failure behavior of virtual machine live migration operations in the wild using data collected over several months of operation at scale. The conclusions derived from this study provides key guidelines for the development of systems that need to employ VM migrations.

(iii) **Speculor** is a capacity multiplexing system designed to provide fault-scalable multiplexing performance and tolerate a much wider class of failures (than dealt with in CCM). Speculor runs on a virtualized infrastructure management stack that exposes speculative versions of VM migration operation. The stack speculatively executes multiple logically equivalent replicas of VM migrations simultaneously across different physical targets like servers, switches and network links with independently failing hardware and software components. Speculor identifies and supplies the stack with different acceptable physical targets based on its service specific knowledge. The virtualized infrastructure management stack also possesses several techniques

to give Speculor complete control over the cost vs. benefit of speculation (policy) while taking care of the implementation and providing robust semantics, leading to a clean separation of concerns. Speculor has been implemented for the open source OpenStack cloud platform running Xen hypervisors.

(iv) **Xerxes** is a distributed resource load generator that can be used to benchmark dynamic capacity allocation and multiplexing systems. It can replay trace utilization data from production datacenters, generate load patterns at varying scales based on profiled load signatures to parametrically explore real and anticipated resource consumption scenarios like flash crowds. The key design property of Xerxes is that it decouples the generation of load at scale from any application logic CPU and memory load generators deployed at each datacenter node or VM that is part of load simulation. The load generators in the individual datacenter nodes are designed to not require any coordination during the course of an experiment to generate an overall load pattern across many machines. This gives Xerxes the ability to tolerate multiple node failures, thereby improving the robustness of the experimentation process beyond what is achievable with applications which are typically much less tolerant to failures. We used Xerxes to test several capacity multiplexing prototypes (discussed below) to identify the limiting factors mentioned previously and to refine our design over several iterations.

1.3 Dissertation Structure

The rest of this dissertation is organized as follows. Chapter 2 discusses the design of Xerxes, the distributed load generation framework and shows several examples of load patterns that can be generated with it.

Chapter 3 presents CCM (Cloud Capacity Manager) and its resource abstractions, algorithms, implementation and evaluation on a 700 server datacenter. The motivation behind its architecture are quantified by analyzing resource utilization

data collected from production datacenters.

Chapter 4 presents a detailed analysis of VM migration costs and failures observed during the development of several early prototypes of CCM, that further influenced the design of its algorithms and fault-tolerance methods. The insights from this study to also motivate the need for Speculor.

Chapter 5 explains in detail the design and implementation of speculative VM migration operation and that of Speculor which employs a two-pass top-k algorithm that emits speculative VM migration actions. We show simulation results validating our hypothesis.

Each of the above chapters also include a detailed survey of salient research related to the systems and topics dealt within them.

Chapter 6 presents our conclusions and discusses future avenues of research and the general applicability of speculative replication of management operations.

CHAPTER II

XERXES: DISTRIBUTED LOAD GENERATOR

An important problem facing developers of capacity allocation and multiplexing systems is the evaluation of these systems at large scales. Existing applications do not operate across thousands of servers without hitting performance bottlenecks due to applications logic, data access limitations (e.g. N-tier applications) [75] or the inability to deal with failures (e.g. HPC applications) [51]. Benchmarking is currently restricted primarily to embarrassingly parallel workloads. Even then, it is difficult to generate appropriate inputs for them that represent realistic cloud scenarios. The net effect of all of these factors to system designers is the danger of narrow design assumptions and optimizations that ultimately lead to poor application performance during deployment.

We develop the *Xerxes* distributed load generation framework to fill this void. We use it to evaluate CCM and Speculor at scale, to understand the behavior of several design choices in a variety of actual or anticipated scenarios. It has enabled us to institute a process of continuous refinement in the development of our systems.

The key property behind Xerxes is that it decouples the generation of load at scale from any application logic. It offers the ability to generate load patterns at both individual node levels, and collectively across a large number of machines. Various interesting load patterns, including large volume spikes [39] across a large number of machines, can be easily generated. Finally, resource usage traces from real-life deployments can also be adapted and replayed in datacenters of varying sizes.

Xerxes is composed as a collection of four independent load generators – one each for CPU, memory, storage and network resources – deployed on independent

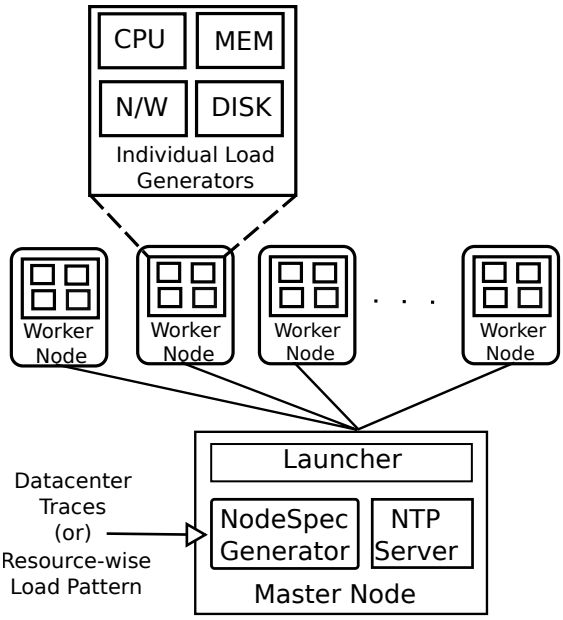


Figure 1: Xerxes architecture.

physical or virtual machines in the datacenter. The load generators in the individual datacenter nodes are designed to not require any coordination during the course of an experiment to generate an overall load pattern across many machines. This gives Xerxes the ability to tolerate multiple node failures, thereby improving the robustness of the experimentation process beyond what is achievable with applications which are typically much less tolerant to failures. The accuracy of the generated load pattern at scale decreases linearly with the number of component node failures.

2.1 Design and Implementation

Figure 1 shows the overall Xerxes framework architecture. Xerxes consists of a single master node and multiple worker nodes - one per server under evaluation. The master node takes load generation specifications, converts them to individual worker node specifications that when executed together produces a global, large scale aggregate load pattern. The individual node specifications are generated by the NodeSpec Generator Module shown in the figure, and their execution is orchestrated by the Launcher Module through the Linux cron facility. The load specifications to the

master can be either in the form of (a) real-life datacenter traces specifying resource usages at various timestamps or (b) statistical distributions, such as normal distribution with a specified mean and deviation values, for example. In addition, these specifications can be at a per-server level or per-logical-job level, that maps to multiple servers, to generate a global resource usage pattern. The base specifications are extrapolated (in the current implementation, only proportionally) when the number of target servers is greater than the number of specification objects in the benchmark input, or, combined via aggregation in the opposite case.

Further, it is also possible to add usage volume spikes to the base load specifications by specifying the spike parameters as characterized by Bodik et. al. [39]: *time-spike-start, time-peak-start, time-peak-end, time-spike-end, spike-magnitude-multiplier*.

In order to orchestrate a global resource usage pattern, the master runs an NTP server that the worker nodes need to synchronize with periodically (typically in days on modern machines), so that their individual timeofday values are not hugely divergent. However, once the simulation starts, the worker nodes need no further coordination with the master and use local high precision timers to transition between multiple load phases.

The worker nodes are composed of four individual load generators: one each for CPU, memory, network and storage resources. The worker node gets individual load specifications (or none, as required) for each generator at the start of the simulation from the master, which it then executes in isolation until completion. The worker load specifications consist of many load phases, one per line, of the form:

$$< period - secs, load - percentage >$$

.

.

$$< period - secs, load - percentage >$$

For each phase in the specification above, the CPU load generator generates

$load - percentage$ by alternating between performing numerical computation over an integer array's elements and sleeping (using Linux nanosleep) at microsecond granularity, many times over until $period - secs$ seconds have elapsed. For example, using a 100 microsecond period and a desired load of 50% would mean that the generator simply computes for 50 microseconds and sleeps for the remaining 50. The CPU generator periodically calibrates itself to determine the number of computations required for a span of 1 microsecond in order to operate in a virtualized environment where the amount of available CPU resource varies with levels of consolidation.

The generator can also be configured to do a fixed amount of work instead. However, from our practical experience on virtualized systems this mode of operation results in decreased accuracy in global load pattern generation.

The memory load generator works similarly to the CPU generator in terms of transition between phases, but interprets the $load - percentage$ as a fraction of a large pre-configured memory size (could potentially be the total available worker node memory size) specified in megabytes. It allocates an integer array buffer of size corresponding to the fraction specified for a particular phase, and performs either random access or linear access of the elements of the array as required for $period - secs$ seconds. Our future work will address the development of the network and storage load generators.

In our current Xerxes prototype the master node components, except the NTP server, are implemented in Python. We use the NTP server available through the Ubuntu software repository. The CPU and memory load generators are written in C for Linux kernel versions 2.6.19 and above where the high-resolution timer API is available.

The following section presents several examples of how the Xerxes framework can be used to create a variety of CPU and memory resource usage patterns in the datacenter.

2.2 Load Generation Scenarios

We first run Xerxes on a 700 node private cloud constructed on a datacenter on campus using the VMware vSphere [27] virtualization platform. Each server has 2 dual core AMD Opteron 270 processors, a total memory of 4GB and two NICs capable of 1Gbps and 5Gbps respectively. The hosts are all connected to each other and a central storage array of 4.2TB total capacity via a Force 10 E1200 switch over a flat IP space. The worker nodes are Ubuntu Linux 9.10 virtual machines, each configured with 4 VCPUs and 1GB of memory, stored on the central storage array. The master node is run on a separate physical server running Ubuntu Linux 9.10 as well. Our monitoring infrastructure is built using the VMware vSphere Java SDK [26] that allows us to fetch resource utilizations samples per-VM once every 20 seconds at small scales and once every 5 minutes at larger scales.

2.2.1 Replaying Datacenter Traces

Recently, Google Inc. released a large scale, anonymized production workload trace from one of its clusters [11, 44], containing data worth over 6 hours with samples taken once every five minutes. Their workload consists of 4 large jobs that each contain a multitude of sub tasks that map to their cluster machines in an unknown way[1]. Each row in the trace presents the resource usage of a single task, belonging to one of the four jobs, at a given timestamp.

The CPU usage is expressed as normalized value of the average number of cores used by the task and the memory usage is expressed as the normalized value of the average memory used by the task over the last 5 minute interval. This provides us sufficient information to replay the resource usage pattern of the four major jobs in the trace on 1600 VMs running on 512 of the 700 servers in our private cloud. For the

[1]We use the trace version 1 where this information was unavailable. A subsequent update from Google provides more detailed information.

sake of simplicity our global load specification evenly partitions the VMs into four sets of 400 VMs where each set is to replay the CPU and memory usage of a unique job.

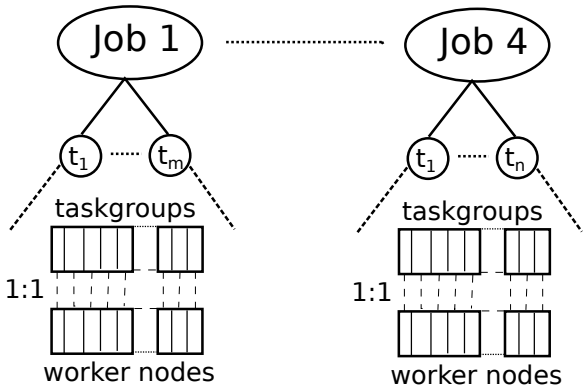
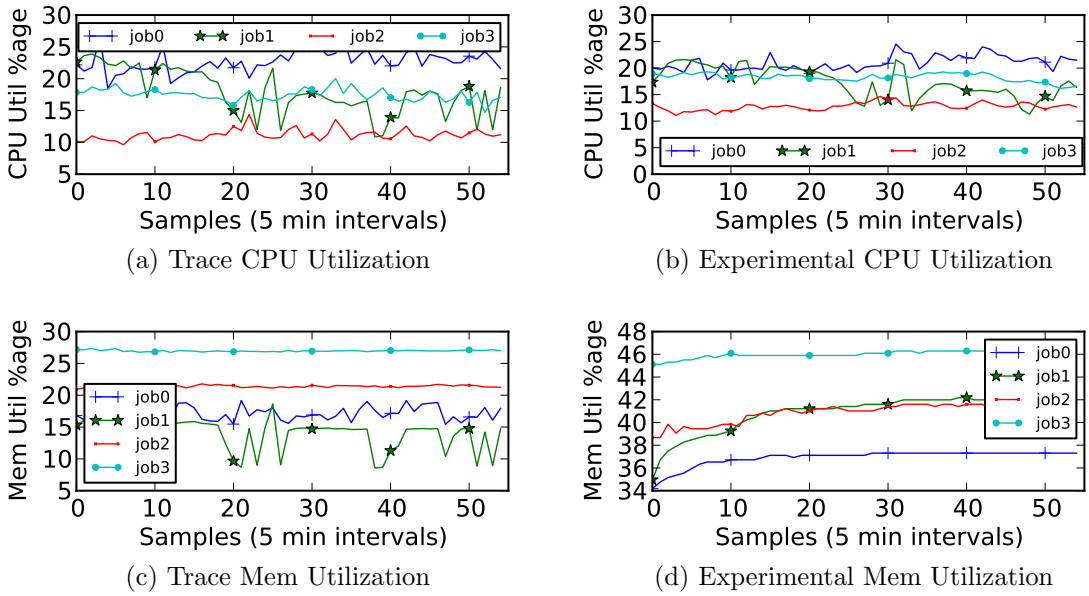


Figure 2: Mapping Google Trace to Xerxes Model.

The number of unique tasks for each job at each timestamp varies over time, with numbers in the tens of thousands, on average. The NodeSpec Generator at the master evenly partitions tasks of a job, at a timestamp, into 400 taskgroups, one taskgroup each for each worker VM as illustrated in Figure 2. We simply re-normalize the resource usage of each task group to a percentage value (assuming a base maximum value) and generate the entire single worker load specification as a series of utilizations to be generated every 5 minutes working up to around 5 hours. Thus each worker node has a different load pattern corresponding to its taskgroup but they together produce the overall global job patterns required.

Figures 3a and 3b show the CPU utilization per job computed from the trace and measured from an actual benchmark run, respectively. Given the scale of the experiment, it can be seen that the overall job load patterns are reproduced fairly accurately in the experimental run. Note that we did observe worker node failures during the course of the experiment and also that there are limits to the granularity of our monitoring setup measuring the utilization in the infrastructure.

Similarly, Figures 3c and 3d show the memory utilization per job computed from



(a) Trace CPU Utilization

(b) Experimental CPU Utilization

(c) Trace Mem Utilization

(d) Experimental Mem Utilization

Figure 3: Job-wise Resource Utilizations for Google Traces.

the trace and measured from an actual benchmark run, respectively. Our datacenter monitoring module measures the memory utilization of entire VMs: this includes the load generator's memory consumption and the VM guest OS utilization resulting in the reported experimental samples being higher than the ones computed from the trace. From our practical observations running the memory load generator, we observed that it is hard to generate accurate memory loads for VMs that have a small amount of configured memory due to the higher fraction of the base memory usage by the guest OS and any other services compared to VMs with large amount of memory (e.g., base = 400MB of 1GB vs. base = 400MB of 10GB).

2.2.2 Replaying Patterns Extracted from Workloads

Xerxes can also be used to replay the resource consumption patterns of applications profiled offline, at varying scales through user-specified extrapolation or contraction functions. To demonstrate this, we first characterize the CPU and memory consumption pattern of four popular cloud applications, each representative of an important class of cloud codes [52]. Briefly, the applications and their configuration are:

Data Analytics: We run a web-search and indexing job in our datacenter using the Nutch [3] search engine that is used to crawl an internal mirrored deployment of the popular Wikipedia.org website containing millions of pages of articles. The local deployment allows us to avoid WAN traffic that would skew the workload characterization results.

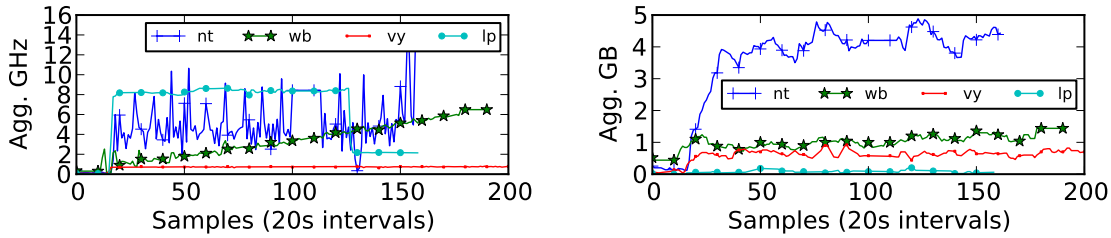
Data Serving: For our characterization of this class of workload, we use the Voldemort [18] key-value store, known for its use at LinkedIn, and drive load to it using the Yahoo Cloud Serving Benchmark [40]. Our workload profile consists of 2 million operations (50% reads and 50% writes) with record request popularity following a zipfian distribution. Each record is 64KB in size.

Web Services: N-tier web applications (usually n=3) form the basis for some of the largest online services. For our characterization, we built a 3-tier, airline reservation benchmark that uses Apache Geronimo for the web server, a Tomcat query processing engine for the middle-tier and a HBase backend instead of a SQL-database, capable of achieving good horizontal scalability. We obtained airline fare data from one of our industry partners, Travelport Inc., as well as request traces numbering in their tens of thousands from a real-life deployment. We modified the httperf tool [13] to generate load by replaying these traces for around an hour using three threads with each thread's requests exponentially distributed with a mean inter-arrival time of 1 second.

High Performance Computing: We use programs from the LAPACK [16] package to solve a system of simultaneous linear equations as a representative workload in this space.

Figure 4 shows the CPU and memory usages of the four target applications. Each application is composed of 5 VMs and the utilization values reported are the aggregate usage of all of the VMs of a given application.

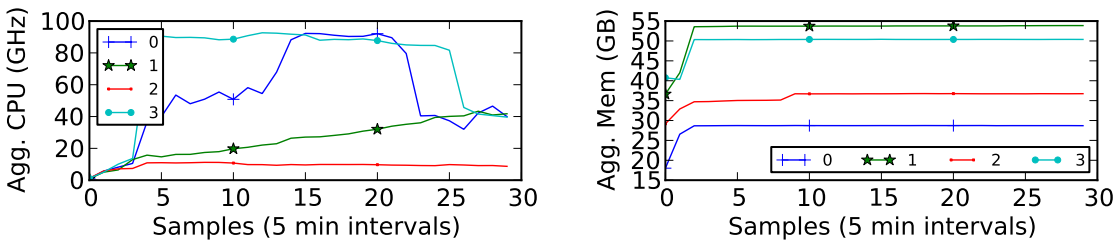
It can be seen that the Nutch application is fairly resource intensive across both



(a) CPU (b) Memory

Figure 4: Aggregate Resource Utilizations of Workloads. Key: nt = Nutch, wb = 3-tier web, vy = Voldemort, lp = LAPACK.

resource types, and that the usage pattern is bursty. In terms of dynamically allocating resources, this presents an interesting tradeoff between dedicating resources to handle the spikes (over provisioning) vs. allocating resources to handle either the average load or a higher percentile (under provisioning). The 3-tier web benchmark has steadily increasing CPU and memory requirements during the experiment as the application becomes more and more backlogged with requests. The LAPACK high performance computing benchmark has a steady, high CPU requirement characteristic of this class of applications. Finally, the Voldemort data serving benchmark appears to only require a small but fairly steady amount of memory resource for its operation. This is due to the fact that we configure the benchmark to store data in-memory and also due to the fact that our record size was only 64KB.



(a) CPU Utilization (b) Memory Utilization

Figure 5: Resource Utilizations for Each Application Profile. Key: 0 = Nutch, 1 = 3-tier Web, 2 = Voldemort, 3 = LAPACK.

Now we take each application's CPU and memory usage profile above and, extrapolate it such that it can run on 400VMs hosted on 128 machines. Recollect that each application profiled has resource usage information of each of its 5 VMs at a 20 second granularity. In our load specification, we simply map the resource usage of each single VM belonging to a given application to 20 VMs in our target simulation to come to a total of 400 VMs. Each application profile will be simulated by 100VMs now. Note that Xerxes has and can be extended to support more complex mappings as well. In addition, to demonstrate the ability of the Xerxes framework to simulate large spikes in the cloud, we added a volume spike to the CPU profile of the Nutch application in the specification. The spike lasts from 60 minutes into the simulation until 120 minutes, and approximately triples the overall workload volume.

Figure 5 shows the results of this scenario. It can be seen that even with the coarse monitoring granularity, the overall CPU usage of each application's VMs is similar to that characterized before. The Nutch profile shows the volume spike, across 100 VMs, as added by Xerxes. However, as with the previous scenario the memory monitoring data is less accurate for the same reasons as above.

2.3 Related Work

To the best of our knowledge, Xerxes is the first publicly available [15] micro-benchmark that gives system developers precise control over the global resource consumption patterns of VMs in a datacenter. The majority of workloads used for benchmarking capacity allocation and multiplexing systems intended for the cloud environment employ scale-out workloads that are stateless and embarrassingly parallel for the most part. These are applications that have been designed explicitly for the cloud and operate on large data sets serving independent user requests and are data locality aware.

Ferdman et.al. introduce a suite of emerging scale-out cloud workloads and characterize their micro-architectural behavior during execution [52]. Sobel et.al. introduce the CloudStone Web 2.0 benchmark that consists of Olio, a social calendaring application, and Faban, a markov-chain based application workflow benchmarking tool [92]. The Olio application typically consists of multiple stateless tiers that access persistent data stored in a MySQL database. Such an architecture is known to hit scaling bottlenecks in the persistence tier [75]. Huang et.al. present HiBench, a collection of synthetic and real-world map-reduce programs, and use it to analyze the performance and power characteristics of Hadoop clusters [64]. Their results show that the resource usage characteristics vary widely based on the hardware platform and software version. YCSB [40] and YCSB++ [86] allow the benchmarking of key-value stores based on different statistical request distributions and request sizes. CloudCmp [72] is a framework to compare the service interface performance of different cloud providers and then use that characterization to predict the performance of legacy applications when deployed in those clouds.

Despite these tools, capacity allocation and multiplexing system designers face two key problems: (i) the application-model similarity in the workloads leads to narrow assumptions and over-optimizing metrics, abstractions and algorithms for a specific case while leaving out workloads like enterprise and scientific applications which form a large fraction of deployed software in IT environments, (ii) systematically exploring the different configuration parameter settings (say Hadoop, for example) to characterize the variation in resource consumption patterns is a challenging task. Babu et.al. show that the performance and resource consumption characteristics of Hadoop applications vary widely based on different settings for the 190 odd Hadoop tuning parameters [36]. They also note that some of these configuration parameter settings interact with one another further complicating the issue.

A second class of benchmarking tools used in the datacenter context are resource

usage trace replay frameworks, similar to Xerxes. Moore et.al. present three tools that aim to automate the collection, analysis and replay of datacenter resource and infrastructure usage traces [80]. However, their sstress tool only replays resource consumption on a single server and is incapable of producing orchestrated global patterns like Xerxes. Delimitrou et.al. emphasize the need to decouple access to applications, their deployment and configuration, to study the I/O performance of a cluster of storage servers, similar in principle to ours [48]. They develop a probabilistic model to characterize the storage performance and access profile of common applications while we rely on traces as input. Further, their system does not allow generation of anticipated scenarios of flash crowds like Xerxes.

2.4 Chapter Summary

In this chapter we argued for the decoupling of scalable load generation from application logic in order to aid researchers/developers test their cloud systems at large scale, which ultimately prevents narrow system design assumptions that ignores the issues seen at scale. We demonstrated the use of a distributed load generation framework, on a 700 node private cloud virtualized with the VMware vSphere virtualization stack, that can: (i) replay real-life datacenter traces, (ii) extrapolate and replay workload characterization data and, (iii) simulate resource usage volume spikes across a large number of machines. In the following chapter we demonstrate how Xerxes helps us identify critical practical design restrictions to iteratively refine our system prototypes.

CHAPTER III

CCM: CLOUD CAPACITY MANAGER

The ability to dynamically multiplex datacenter compute capacity amongst workloads with time varying demands allows datacenter administrators to oversubscribe resources, compared to configured VM capacities or worst-case demands (peak demand), and leads to significant operational efficiency. In this chapter we introduce a system called CCM - Cloud Capacity Manager - that solves the problem of doing such *capacity multiplexing at large scales* across an entire datacenter.

Across an entire datacenter, the monitoring of physical and virtual entities at fine granularities poses significant network overhead. This is especially acute in the common bandwidth oversubscribed tree-based datacenter network architecture [34]. Much of the existing prior art on capacity multiplexing, even those intended for large scale deployments, tend to focus on the development of accurate methods for workload demand prediction and allocation, and, application service level agreement (SLA) compliance [85, 56, 79, 100, 96, 90]. Such methods inherently require fine grained monitoring information in order to function.

In a similar vein, the cost of actuation, i.e., performing VM migrations to dynamically re-allocate resources, at large scales is fairly expensive [94] (again, exacerbated, given bandwidth oversubscribed networks) and non-deterministic [93] in terms of cost and time. Therefore, there are also limits to the amount of actuation that can be done to multiplex capacity at various time bounds, across an entire datacenter. Therefore, the accuracy of allocation methods matters far less than what most research prototypes usually assume.

We take these observations into consideration in the design of CCM as follows.

CCM achieves reductions in monitoring and actuation cost at scale by employing a 3-level hierarchical management architecture. The capacity managers at each level continuously monitor and *aggregate black-box VM CPU and memory usage* information and then use this aggregated data to make *independent* and localized capacity allocation decisions. Data aggregation reduces the amount of information that flows across the hierarchy. In addition, monitoring and resource changes occur at progressively less frequent intervals when moving up the hierarchy, i.e., at lower vs. higher level managers, the reasons for which are explained later in detail.

The core concept embodied in CCM's capacity multiplexing is the on-demand balancing of load, across logical host-groups, at each level in the hierarchy. Keeping in mind that the cost of actuation dominates at scale, CCM eschews complex workload demand forecasting and allocation methods, and uses simple greedy hill-climbing algorithms to iteratively reduce load imbalances across the hierarchy. It also uses management operation throttling and liberal timeouts, to select cheaper operations, and, minimize the incidence of operation failures due to resource insufficiency as explained in detail later. Our overarching design principle with these techniques is *simplicity*, much in line with suggested best practices for developing large scale cluster systems [62, 65]. Such simple methods are easier to scale, develop and debug, and, we demonstrate that they do not sacrifice on effectiveness through detailed large scale experiments.

CCM answers the question of not only scalably multiplexing datacenter capacity, but also doing it *safely* i.e., while capturing and ensuring that VMs demands are met appropriately. Therefore, in addition to allocating capacity purely based on VMs' runtime resource demands, CCM also offers absolute minimum, maximum and proportional resource allocation QoS (quality of service) controls for each VM, interpreted across the entire cloud.

In order to properly characterize CCM's scalability and resilience to the aforementioned practical issues in real-life environments, we deploy and evaluate CCM on a 700 physical host datacenter, virtualized with the VMware vSphere [27] virtualization platform. We generate realistic datacenter load scenarios using the Xerxes distributed load generator discussed in the last chapter and additional application workloads. The application workloads include CloudStone [92] – a 3 tier web 2.0 application, Nutch [3] – a map-reduce based crawler, Voldemort [18] – a key-value store and the Linpack high performance computing benchmark [12].

In the remainder of this chapter, Section 3.1 presents background on the vSphere QoS abstractions that can be used to achieve desired isolation and sharing objectives at smaller scales. These abstractions are adopted and extended by CCM in the cloud context. Section 3.2 presents CCM's black-box metrics computed for dynamically allocating capacity to application VMs at cloud scale, the assumptions we make, and its load balancing algorithms. Section 3.3 outlines CCM's implementation challenges, with a focus on methods for dealing with management failures. Section 3.4 presents a detailed experimental evaluation. Related work in Section 3.5 is followed by a summary of conclusions from this chapter.

3.1 vSphere Basics

As stated previously, CCM is built on top of the VMware vSphere [27] is a datacenter virtualization platform that enables infrastructure provisioning and virtual machine lifecycle management. In particular, CCM relies on VMware vSphere platform features that include (i) Distributed Resource Scheduler (DRS) [25] (ii) Distributed Power Management (DPM) [24]. These features are employed at the lower levels in the CCM hierarchy for management of smaller scale clusters of hosts. Hosts are arranged into logical clusters, each a collection of individual physical hosts. The overall resource capacity of a cluster is the sum of all individual host capacities. With

vSphere, a central server offers these features for a cluster of at most 32 hosts. We refer the reader to [61] for a detailed discussion of the design of DRS and DPM.

DRS Capacity Allocation. DRS automates the initial and continuing placement of VMs on hosts based on the following resource QoS controls.

Reservation (R): a per resource minimum absolute value (in MHz for CPU and MB for memory) that must always be available for the corresponding VM. To meet this requirement, VM admission control rejects VMs with reservations when the sum of powered on VMs' reservations would exceed total available cluster capacity.

Limit (L): a per resource absolute value denoting the maximum possible allocation of a resource for the VM; strictly enforced even in the presence of unused capacity.

Share (S): a proportional weight that determines the fraction of resources a VM would receive with respect to the total available cluster capacity and other resident VMs' shares; allows graceful degradation of capacity allocation under contention.

Details on how to choose appropriate values for these controls, based on application QoS needs, appear in [7]. For each VM, DRS computes a per-resource *entitlement* value as a function of its own and other VMs' resource controls and demand, and, also the total available cluster resource capacity. VM demand is estimated based on both its present and anticipated future resource needs: computed as its average resource usage over a few minutes plus two standard deviations. Using the entitlement values, DRS computes a set of VM to host associations and performs migrations as necessary.

The CCM algorithms, described in the following section, extend the interpretation and enforcement of these resource control abstractions to cloud scales of thousands of hosts.

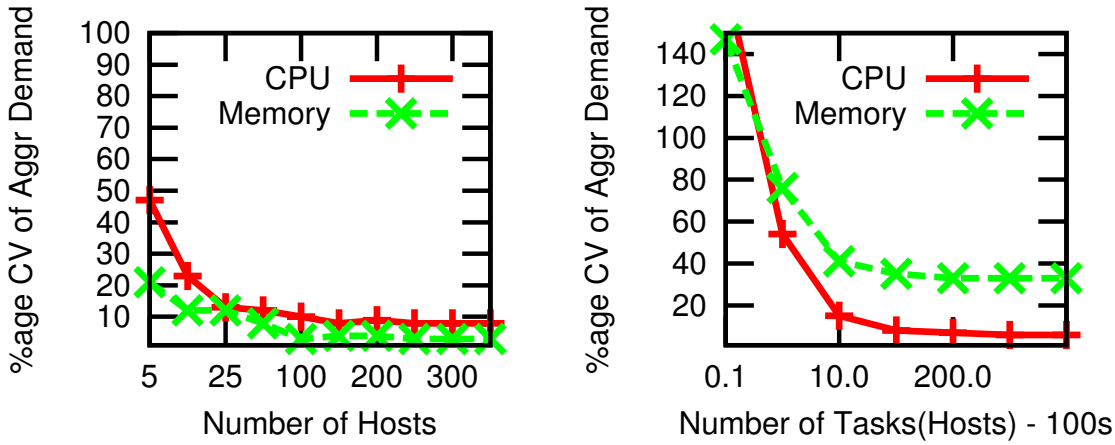
DPM Power Management. Whenever a cluster host's utilization value falls below a specified threshold, DPM performs a cost benefit analysis of all host power downs in the entire cluster, so as to raise the host utilization value above the threshold by accepting the VMs from the powered down hosts [61]. The outcome of this analysis

is a per-host numeric value called *DPMScore*. A higher DPMScore denotes greater ease with which a host can be removed from operation in a cluster. CCM uses this value to aid in host re-association between clusters and superclusters, as explained next.

3.2 Design

It is well-known that clustering and hierarchies can help with scalability [71] by reducing the overheads of system operation. More importantly, in the capacity allocation context, as resource consumption of workloads is aggregated across larger and larger numbers of physical machines, as one moves upwards from lower levels in the hierarchy, there is the possibility of decreased degrees of variation in this aggregate consumption. The intuition here is that across a large set of servers in a typical multi-tenant cloud, the individual customer workload resource consumption patterns are not likely temporally correlated, i.e., their peaks and valleys do not coincide. As a result, there could be substantial opportunity for workload multiplexing, which increases with the size and the diversity of the workload [79]. *In terms of its system design implication, this permits monitoring and management operations to achieve low overhead by running at progressively larger time scales, when moving up the hierarchy, without substantive loss in the ability to meet changing workload resource demands.*

We reinforce this idea by analyzing the aggregate resource consumption in two production datacenter traces from: (a) an enterprise application cluster running the VMware vSphere [27] datacenter virtualization platform and (b) a publicly available trace from Google's production cluster [11] (version 1). Figure 6 presents the coefficient of variation (as percentage) in aggregate CPU and memory consumption across larger and larger groupings of physical hosts, in each trace. The graphs show that the coefficient of variation, for both CPU and memory resources, decreases roughly for



(a) VMware vSphere Cluster (b) Google Production Cluster

Figure 6: Variation in Aggregate Demand over increasing scales from production traces.

each order of magnitude increase in the number of hosts considered for aggregation.

The above factors, therefore, motivate the natural fit of a hierarchical management architecture to capacity allocation. Hierarchies can also be defined so as to a) match underlying structures in the datacenter in terms of rack boundaries, performance or availability zones, different machine types, etc., or to b) capture structure in the application such as co-locating frequently communicating VMs in the same part of the infrastructure. We plan on exploring this in our future work.

The CCM architecture, shown in Figure 7, is organized as a hierarchy of a top-level cloud manager, mid-level supercluster managers, and finally cluster managers at the lowest level. Per-VM management takes place only at cluster level. At this level, VMware DRS is used to make independent and localized decisions to balance loads as briefly explained in Section 3.1. At higher levels, i.e., supercluster and cloud, CCM computes the total estimated demand of a cluster or supercluster, respectively, as an *aggregation* of per-VM resource usage, i.e., the total demand of all VMs running under the given lower-level management entity in the hierarchy. This total demand estimate is then used to determine the amount of capacity required by a cluster or supercluster and to perform coarse grain capacity changes, as necessary. The capacity

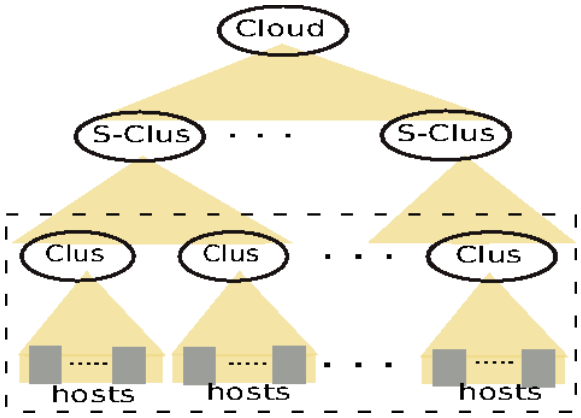


Figure 7: Hosts are grouped into a logical cluster with a cluster-level capacity manager (VMware DRS), clusters are grouped into a supercluster with a corresponding capacity manager and a cloud is composed as a collection of superclusters under a cloud level capacity manager.

changes are realized by *logically re-associating* hosts between cluster and supercluster managers. A logical host re-association or *a host-move operation* is a two step process where the VMs running on a chosen host are first live migrated to other hosts in the *original source cluster* and then the source and destination managers' inventories are updated to reflect the change. This operation, however, preserves VM-to-capacity-manager associations across the hierarchy. The benefit of this property and the rationale behind its choice are explained in detail in Section 3.2.1. Such coarse-grained higher level capacity changes, then, automatically trigger the cluster-level re-balancing actions of DRS. Finally, for reasons cited above, both the monitoring and resource changes in our system occur at progressively less frequent intervals when moving up the hierarchy.

Metrics. The following key metrics are computed and used by CCM algorithms at the supercluster and cloud levels of the management hierarchy to determine the capacity to be allocated to these entities, based on their aggregate demand and R, L, S constraints.

Entitlement(E): For each VM, per-resource entitlements are computed by DRS

(see Section 3.1). We extend the notion of entitlement to entire clusters and superclusters, to express their aggregate resource demand. There are no R, L, S parameters for clusters and superclusters. This entitlement calculation is explained in detail in Section 3.2.3.

For the CCM hierarchy, an aggregate notion of demand is computed by successively aggregating VM-level entitlements for clusters and superclusters. Upper-level capacity managers, then, allocate capacity between clusters and superclusters using such aggregate information instead of considering individual VMs, while lower-level cluster managers continue to independently decide and perform detailed allocations.

We note that when aggregating entitlements, some additional head room is added, worth the last two standard deviations in resource usage, so as to absorb minor transient usage spikes and also give VMs the ability to indirectly request more resources from the system, by consuming the extra head-room over longer periods. This results in the successive addition of some amount of excess resource to the computed entitlement as aggregation proceeds up the hierarchy. The effect is that even with coarser sample/estimation intervals, there is a high likelihood of being able to cover a cluster's or supercluster's varying resource demands. The resource volume used for this head-room is directly proportional to the amount of variability in resource demands – i.e., when demands are stable, less resources are added, and vice versa.

Normalized Resource Entitlement(NE): is a measure of the utilization of available capacity. For a given entity, e.g., cluster or supercluster, it is defined as the sum of the total amount of a resource entitled to that entity, divided by the entity's total resource capacity. Thus, NE captures resource utilization, ranging from 0 to 1.

Resource Imbalance(I): captures the skew in resource utilizations across a set of similar entities at the same level in the resource management hierarchy. It is defined as the standard deviation of individual normalized resource entitlements (NE) of the entities in consideration. High I and NE values suggest the need for capacity

reallocation between entities to better back observed VM demand.

DPMRank: is a cluster-level aggregation of the individual host DPMScore values. A high DPMRank metric for a cluster indicates its suitability to cheaply donate hosts to other overloaded clusters, or superclusters in the event of a higher layer load imbalance. It is computed as the squared sum of individual host DPMScores, so that the metric favors clusters with a small number of hosts with very low resource utilization vs. those that have large numbers of relatively modestly utilized hosts. For example, a cluster of 3 hosts with a DPMScore of 10 each will be favored over a cluster of 10 hosts with DPMScores of 3. The reason, of course, is that the cost of re-associating a host with DPMScore of 10 is less than that of a host with DPMScore of 3.

3.2.1 Logical Capacity Re-association

As mentioned before, CCM logically re-associates capacity between cluster and supercluster managers. We chose such re-association based on several practical considerations. First, in any given large scale system, hosts fail and recover over time, causing these systems to incorporate capacity changes in their design, thereby making dynamic host addition and removal a standard mechanism. Second, VM migration would also require moving substantial state information about each such VM accumulated at the corresponding capacity managers. This includes statistics computed about the VM's resource usage over the course of its existence, runtime snapshots for fail-over, configuration information, etc. Migrating this state along with the VM to the new capacity manager, would be costly, and it would potentially seriously prolong the delay experienced until the VM and capacity management once again become fully operational. While we do have mechanisms to migrate VMs across clusters, we found host re-association to be a simpler solution from a design and implementation standpoint.

For the reasons above, CCM preserves the association of VMs to clusters and superclusters, and it manages capacity by logically re-associating *evacuated hosts* between them. Host re-association uses the DPMRank value to identify clusters with hosts for which evacuation costs are low, and once a host has been moved to its new location, the lower level resource managers (DRS/ESX) notice and automatically take advantage of increased cluster capacity by migrating load (i.e., existing VMs) onto the new host. In this fashion, CCM achieves a multi-level load balancing solution in ways that are transparent to operating systems, middleware, and applications.

3.2.2 Assumptions

In the construction of CCM, we make two basic assumptions about datacenters and virtualization technologies. (1) Hosts are assumed uniformly compatible for VM migration; this assumption could be removed by including host metadata in decision making. (2) Storage and networking must be universally accessible across the cloud, which we justify with the fact that there already exist several instances of large scale NFS deployments, and VLANs that are designed specifically to facilitate VM migration across a large pool of hosts. Further assistance can come from recent technology developments in networking [83, 57, 22], along with the presence of dynamically reconfigurable resources like storage virtualization solutions, VLAN remapping, switch reconfiguration at per VM level, etc. Finally, these assumptions also let us carefully study the impact of management operation failures and costs on system design.

3.2.3 Capacity Management Algorithms

Table 1 summarizes the metrics described previously. CCM's cloud-scale capacity management solution has three primary allocation phases: (i) Initial Allocation, (ii) Periodic Balancing, and (iii) Reactive Allocation. The initial allocation recursively distributes total available cloud capacity among superclusters and clusters based solely on the underlying VMs' static resource allocation constraints: Reservation,

Table 1: Resource Management Metrics Across All Levels in the Hierarchy. Key: R = Reservation, L = Limit, S = Share, Cap = Capacity, H = Host, C = Cluster, SC = SuperCluster and CL = Cloud. $Metric_X^Y$ denotes value of "Metric" for an entity at level "X" computed at an entity at the next higher level "Y". $Metric_X$ implies "Metric" for an entity at level "X" computed at the same level. I_X denotes the Imbalance metric computed across all sub-entities of level "X".

L	**Entitlement**	**Norm. Ent**	**Imbalance**
C	$E_{VM} \leftarrow f(R, L, S, Cap(CL))$ $E_C = \sum E_{VM}$	$NE_H = \frac{\sum_H E_{VM}}{Cap(H)}$	$I_C = \sigma(NE_H), \forall H$
SC	$E_C^{SC} = E_C + 2 * \sigma(E_C)$ $E_{SC} = \sum E_C^{SC}$	$NE_C = \frac{E_C^{SC}}{Cap(C)}$	$I_{SC} = \sigma(NE_C), \forall C$
CL	$E_{SC}^{CL} = E_{SC} + 2 * \sigma(E_{SC})$	$NE_{SC} = \frac{E_{SC}^{CL}}{Cap(SC)}$	$I_{CL} = \sigma(NE_{SC}), \forall SC$
Cluster		$DPMRank_C = \sum (DPMScore_H)^2$	

Limit and Shares. Once this phase completes, the periodic balancing phase is activated across the hierarchy; it continually monitors and rectifies resource utilization imbalances, i.e., the presence of high and low areas of resource contention. Finally, in order to deal with unexpected spikes in resource usage, CCM uses an additional reactive allocation phase, which is triggered whenever the resource utilization of an entity exceeds some high maximum threshold (e.g., 80%). Reactive allocation quickly allocates capacity using a tunable fraction of idle resources set aside specifically for this purpose. In this manner, the three phases described allow for the complete automation of runtime management of datacenter capacity.

(i) Initial Allocation. The amount of a resource to be allocated to a cluster or supercluster is captured by the *entitlement* metric. For an initial allocation that does not yet have runtime resource demand information, the entitlement value is computed using only static allocation constraints. As capacity allocation proceeds across the cloud, three types of VMs, in terms of resource allocation, will emerge: (a) Reservation-Based VMs (R-VMs), (b) Limit-Based VMs (L-VMs), and (c) Share-Based VMs (S-VMs). A R-VM requires more of a given resource due to its specified resource reservation being larger than the amount of resources it would have been

allocated based only on its proportional shares, i.e., $E_{VM} = R_{VM}$. Similarly, a L-VM requires less of a given resource due to its resource limit being lower than the amount of resources it would have been allocated based on its proportional shares alone, i.e., $E_{VM} = L_{VM}$. The VMs that do not fall into either category have their capacity allocations determined by their Shares value – S-VMs.

For example, consider four VMs A, B, C, and D contending for a resource, all with equal shares. The amount of resources they would receive based on their shares alone is 25% each. Now, if A has a reservation of 40% for the resource and B has a limit of 20%, the resulting allocation for VMs A and B would be 40% and 20%, respectively. VM A, then, is a R-VM, whereas VM B is a L-VM. VMs C and D are S-VMs, and they share the remaining resources (after A's and B's constraints have been enforced) at 20% each.

Therefore, R-VMs reduce the total amount of resources available to be proportionally shared between S-VMs, and L-VMs have the opposite effect. The process of initial allocation, then, involves the identification of the three classes of VMs and then computing the final entitlement values. Across the cloud, R-VMs would be entitled to their reservation values, L-VMs would be entitled to their limit values, and the remaining capacity is used to compute the entitlement for S-VMs.

In order to explain the resource entitlement calculation for S-VMs, we first define a metric for these VMs, called *entitlement-per-share (EPS)*, as ρ such that, $\rho = E_{VM}/S_{VM}$. Since a VM's resource proportion is computed using its share with respect to the share values of *all* of the other VMs in the cloud, the EPS value stays the same for every Share-Based VM. Now, a given S-VM's entitlement can be computed as $E_{VM} = \rho * S_{VM}$. Thus, the complete equation to compute the entitlement of a specific resource of a VM is given by Equation 1.

The goal of initial capacity allocation is to ensure that *all of the available capacity* in the cloud is allocated to VMs in lieu of their allocation constraints as in Equation

2. We simplify the determination of capacity to be allocated to VMs, i.e., entitlement, via a recursive allocation algorithm. The algorithm performs a binary search over all possible assignments for ρ such that Equation 2 is satisfied throughout the process of computing entitlements using Equation 1. VM-level entitlements are aggregated across levels of the hierarchy. There can only be one unique value for ρ that ensures that this sum exactly equals the total available resource capacity of the cloud [60].

$$E_{VM} = MIN(MAX(R_{VM}, S_{VM} * \rho), L_{VM}) \tag{1}$$

$$\sum E_{VM} = Capacity(Cloud) \tag{2}$$

$$E_{VM} = MIN(MAX(R_{VM}, MIN(S_{VM} * \rho, D_{VM}))), L_{VM}) \tag{3}$$

(ii) Periodic Balancing. As mentioned before, the periodic balancing algorithm typically runs increasingly infrequently at lower vs. higher levels. Since the granularity of this interval impacts the overhead and accuracy of the CCM capacity balancing solution, administrators are given the ability to configure the resource monitoring and algorithmic intervals to individual deployment scenarios. The VM entitlement calculation now uses estimated runtime resource demand (D_{VM}) information together with the resource controls as shown in Equation 3. Load balancing between different hosts in a cluster is provided by the DRS software.

For a given set of clusters or superclusters, when the maximum value of the normalized resource entitlement of the set, for a given resource, exceeds an administrator-set threshold and, simultaneously, the resource imbalance across the set exceeds a second administrator-specified threshold, the CCM capacity balancing algorithm incrementally computes a series of host re-associations across the set to try to rectify the imbalance, up to a configured upper limit (explained in Section 3.3). The first condition ensures that CCM does not shuffle resources unnecessarily between entities of

the same level in the hierarchy when the overall utilization is low although the imbalance is high. Once it has been decided to rectify the imbalance, hosts are moved from entities with lowest normalized resource entitlements (and higher DPMRanks) [1] to those with highest normalized resource entitlements. This results in the greatest reduction in resource imbalance. When removing hosts from an entity, we always ensure that its capacity never goes below the amount needed to satisfy running VMs' sum of reservations (R). When selecting particular hosts to be moved from a cluster or supercluster, hosts that have the least number of running VMs are preferred.

(iii) Reactive Allocation. In order to deal with unexpected spikes in resource usage, CCM uses an additional reactive allocation phase, which is triggered whenever the resource utilization of an entity exceeds some high maximum threshold (e.g. 80%). It is typically invoked in the space of a few minutes, albeit progressively infrequently (up to an hour at the cloud level), as one moves upwards in the management hierarchy. But, as seen in Section 3.2, macro-level spikes across larger and larger groupings of physical hosts are increasingly unlikely to occur, especially at small time scales. To aid in the quick allocation of hosts to resource starved clusters, we maintain a per supercluster *central free host pool* that holds a tunable fraction of DPM module recommended, pre-evacuated hosts from across clusters that belong to the supercluster. This also removes the need for the host selection algorithm to be run. CCM currently only holds otherwise idle resources in the central free host pool, but simple modifications to the scheme could allow holding hosts even when the system is being moderately utilized. If there are no hosts in the free host pool or if the hosts currently present are insufficient for absorbing the usage spike, the periodic algorithm has to perform the remaining allocation in its next round.

There are some notable differences between the reactive algorithm running at the

[1] In the actual realization, we sort in descending order of the sum $0.5 * (1 - NE) + 0.5 * NormalizedDPMRank$. Both normalized resource entitlement and NormalizedDPMRank are given equal weights.

supercluster manager vs. the cloud manager. The cloud manager pulls from and deposits to the per supercluster central free host pool as opposed to having to pick a specific sub-cluster. This optimization serves to reduce overhead and will not affect the operation of the supercluster manager. The supercluster manager only maintains the administrator specified fraction of hosts in the free host pool and the rest are automatically distributed among clusters.

3.3 Prototype Implementation

CCM is implemented in Java, using the vSphere Java API [26] to collect metrics and enforce management actions in the vSphere provisioning layer. DRS is used in its standard vSphere server form, but for DPM, we modify the vSphere server codebase to compute the DPMRank metric and export it via the vSphere Java API. For simplicity of prototyping and algorithm evaluation, both the cloud manager and the supercluster manager are implemented as part of a single multithreaded application run on a single host.

In order to improve the efficacy of CCM, an important element of CCM's implementation is the need to deal with non-determinism in operation resource costs and completion times, as well as to handle failures of management operations. Note that the resource cost of an operation is directly proportional to the amount of time it takes. In addition to performing VM migrations within a cluster, a *host-move* is one of the basic management operations performed by CCM at the supercluster and cloud levels, the purpose being to elastically allocate resources across the datacenter.

Figure 8 shows the average time taken for a single host-move operation for a particular run of the experiment in Scenario I in Section 3.4. It can be seen that the duration varies between a wide range of 44 seconds to almost 7 minutes. This fact complicates the amortization of management overhead subject to the workload benefits being derived.

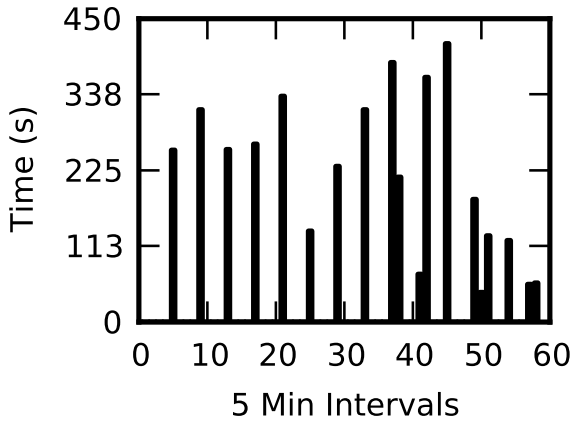


Figure 8: Avg Single Host-move Latency.

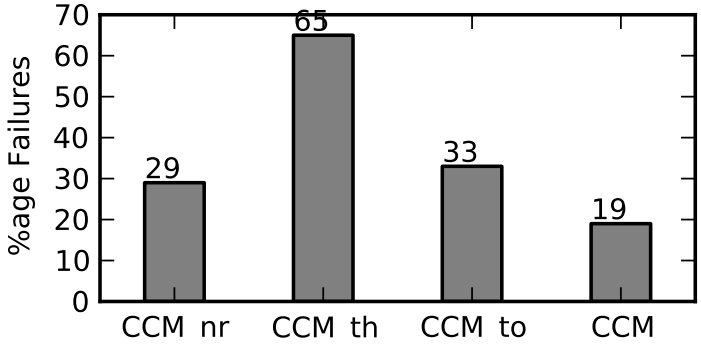


Figure 9: Migration failures.

In addition to the variable cost of management operations, we've also observed significant number of VM live migration failures, between 19% to 65%, at scale, for different experimental runs with different configurations of CCM (explained below), as shown in Figure 9. Note that failure to migrate away VMs would also result in failure of the compound host-move operation leading to sub-par capacity multiplexing. We delve into the details of the causes of VM migration failures next in Chapter 4. A significant portion of the collected error reports suggested a link between failures and higher live migration load subject to infrastructure resource availability (e.g., network bandwidth, CPU at source, etc.) i.e., *higher management resource pressure*.

CCM uses several methods to address the above issues, with a focus on *simplicity*, so as to be practicable at large scales. In order to contain management resource

Table 2: Management effectiveness. Key: NR - No Restrictions, TH - Only Host-move Thresholds, TO - Only Host-move Timeouts, All - TH + TO.

Metric	**NR**	**TH**	**TO**	**All**
Total Host-move Mins	4577	703	426	355
Successful Host-moves	14	24	30	37
emat (moves/hr)	0.18	2.05	4.23	6.25

pressure at any given time, and to avoid the failures seen in Figure 9, we offer administrators the ability to tailor the system's management action enforcement aggressiveness using: (a) explicit throttling of management enactions, i.e., the number and parallelism of host-moves during each periodic balancing round, and (b) automatically aborting long running actions using timeouts, with partial state-rollbacks for consistency. Aborting long running management operations helps to avoid inflating the management overhead and taking away useful resources from applications. It also permits the potential exploration of alternative, cheaper actions at a later time. This also leads to effective management, as indicated by the results shown in this chapter, without demanding expensive fine-grained monitoring of workload VMs at large scales, otherwise required if trying to proactively predict their migration costs [74].

Finally, the management algorithms at each manager are not employed when there is insufficient monitoring information. Transient monitoring data "black-outs", although rare, happen in our environment due to temporary network connectivity issues, monitoring module crashes, etc. Also, failed multi-step operations use asynchronous re-tries for some fixed number of times before declaring them to have failed.

We now quantify the effectiveness of the simple host-move timeout and thresholding abstractions by turning them on one-by-one in the CCM prototype, resulting in four different system configurations. Table 2 briefs these configurations and shows the total amount of time spent in making all host-moves (including failed ones, counting parallel moves in serial order) and the number of successful host-moves, in each case. The workload and configuration parameters used in the experiment are described in

detail in Scenario I in Section 3.4.

It can be seen that using a combination of limiting host-moves and using abort/retry, CCM exhibits higher management action success rates and lower resource costs (directly proportional to the total enforcement time) in our datacenter environment. This leads us to a direct formulation of a new management goodput metric – *effective management action throughput (emat)* – with which it is possible to more objectively compare the different configurations:

$$emat = \frac{num\_successful\_host\_moves}{total\_time\_spent\_in\_making\_ALL\_moves} \tag{4}$$

The emat metric, shown in the final row of Table 2, sets a two- to six-fold advantage of using both thresholding and abort/retry, over the other configurations.

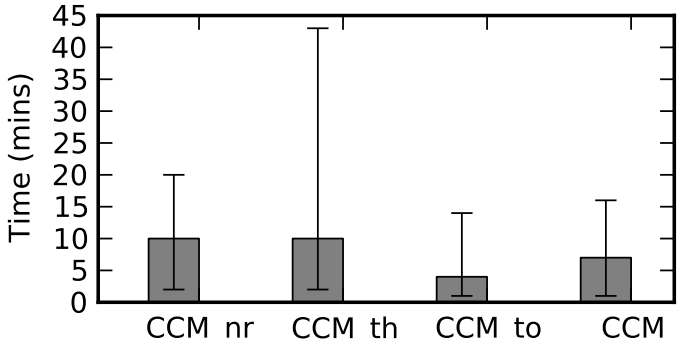


Figure 10: Host-move times

As stated previously, in addition to the host-move failures, it is also important to take into account the cost-to-benefit seen for the management actions being enforced. Figure 10 depicts the average, minimum and maximum values of successful host-move action times for all of the four configurations. Given that the resource cost of an action is directly proportional to the amount of time it takes, it is important to abort overly long-running host-moves, and this can be seen from the steadier host-move times, observed with CCM_to and CCM, each of which are more likely to have a favorable cost-to-benefit.

These results underscore the limitations imposed by practical issues in large scale setups and how simple methods that *explicitly design for these restrictions* can lead to better overall capacity allocation outcomes.

3.4 Experimental Evaluation

The experimental evaluations described in this section show: (i) that CCM is effective in keeping a cloud load balanced, by shuffling capacity based on demand; (ii) this reduces resource consumption hotspots and increases the resources available to workloads, and as a consequence, improves overall datacenter utilization; and (iii) CCM incurs operational overheads commensurate with the dynamism in the total workload. We measure system overhead as the number of VM migrations performed. Migrations are the primary and biggest contributor, in terms of resource usage and time, to the cost of capacity management. The numbers reported in this section are averaged over 3 runs of each experiment for the first two scenarios and 4 runs for the final scenario. We furnish information on the variability in performance where it is non-trivial (i.e., $> 1\%$).

In the absence of a datacenter-wide capacity multiplexing solution, administrators typically resort to statically partitioning their servers and employing traditional capacity multiplexing solutions within each partition [31, 29]. This strategy, which we refer to as partitioned management (PM), forms the basis for comparing CCM's performance and overheads. We emulate such a strategy by using VMware DRS inside each partition to continuously migrate VMs amongst the machines in the partition in response to load changes. CCM, then, naturally builds upon and extends this strategy with techniques explicitly designed to deal with issues at scale. This makes it easier to adopt in existing deployments.

Testbed and Setup: CCM operates on a private cloud in a 700 host datacenter. Each host has 2 dual core AMD Opteron 270 processors, a total memory of 4GB and

runs the VMware vSphere Hypervisor (ESXi) v4.1. The hosts are all connected to each other and 4 shared storage arrays of 4.2TB total capacity via a Force 10 E1200 switch over a flat topology. The common shared storage is exported as NFS stores to facilitate migrating VMs across the datacenter machines. The open-source Cobbler installation server uses a dedicated host for serving DNS, DHCP, and PXE booting needs. VMware's vSphere platform server and client are used to provision, monitor, and partially manage the cloud.

Two important infrastructure limitations in this testbed influence our experiment design: (i) each server has a fairly low amount of memory compared to current datacenter standards, so that over-subscribing memory has non-negligible overhead [95], and (ii) each server has bandwidth availability of approximately 366 Mbps during heavy use, due to a relatively flat but somewhat under-provisioned network architecture, able to efficiently support VM migration only for VMs configured with moderate amounts of memory. As a consequence, most experiments are designed with higher CPU resource vs. memory requirements. Capacity allocation with respect to one resource proceeds as long as the imbalance or maximum utilization of other resource(s) do not cross their respective thresholds.

The private cloud used in experiments is organized as follows. The total number of hosts is divided into 16 partitions in the PM case, with an instance of VMware DRS managing each partition. The DRS instances themselves run external to the partitions, on four individual hosts, as part of corresponding vSphere server instances. CCM builds on these base partitions or clusters by managing each set of 4 clusters via a supercluster manager (4 total). All of the supercluster managers, in turn, come under the purview of a single cloud level manager. The cloud level and supercluster level managers of CCM are deployed on a dedicated physical machine running Ubuntu Linux 9.10. Figure 7 from Section 3.2 shows the overall logical organization of both PM and CCM. The organization for PM appears inside the dashed rectangle.

In terms of the cloud-wide QoS controls settings exposed by CCM, we predominantly use the same proportional Shares for all the VMs in our environment with no Reservation and Limit, unless otherwise noted in specific circumstances in Scenario III. The use of high Reservation and Limit settings on the VMs would reduce the flexibility of dynamically multiplexing capacity for both the CCM and PM cases. Since our goal is to study the multiplexing aspect of system design, we use only proportional Shares.

3.4.1 Workloads

Trace-driven Simulation: As discussed in the previous chapter, the Xerxes distributed load generator produces global, datacenter-wide CPU and memory usage patterns. The master takes in a global load specification, translates it to per-VM load specification, and sets up the simulation. Once the simulation starts, the individual VM generators need no further intervention or coordination. We use this tool to replay real-life datacenter traces and also generate global resource usage volume spikes, as will be explained in detail later.

Cloud Application Suite: four distributed applications represent the most commonly deployed classes of cloud codes: (i) Nutch (data analytics) [3], (ii) Voldemort (data serving) [18] with the YCSB [40] load generator, (iii) Cloudstone (web serving), and (iv) HPL Linpack (high performance computing) [12]. The Nutch instance crawls and indexes a local internal mirrored deployment of the Wikipedia.org website, so that we avoid any skews in results due to WAN delays. The crawl job is set to process the top 200 pages at each level up to a depth of 4. The Cloudstone Faban workload generator is modified to only generate read-only requests, in order to avoid known MySQL data-tier scalability bottlenecks [41]. We set Faban to simulate a total of 12k concurrent users. For YCSB, we use 4MB records and a workload profile that consists of 95% read operations and 5% update operations with zipfian request

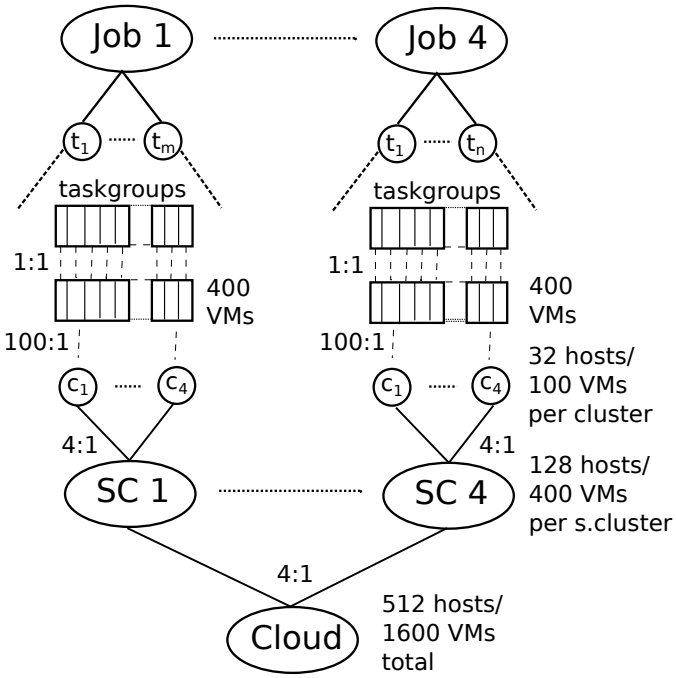


Figure 11: Google trace replay.

popularity. Finally, Linpack solves 8 problems of pre-determined sizes over the course of each experiment, measuring the average throughput achieved in each case.

3.4.2 Scenarios

I. Rightsizing the Cloud: We use the load scenario described in Chapter 2 using the publicly Google cluster traces for this experiment. Using Xerxes, we replay the resource usage pattern of the four major jobs in the trace on 1600 VMs, 400 per job, running on 512 of the 700 servers in our private cloud. All the VMs have equal Shares with no Reservation and Limit control values. For the sake of convenience, we replay only the first 5 hours worth of data from the traces.

Figure 11 illustrates how the trace is mapped to the CCM datacenter: the tasks of a particular job are evenly partitioned, at a given timestamp, into 400 taskgroups, with one taskgroup mapped to a single VM. Across multiple runs of the scenario, for both PM and CCM, the same taskgroup to VM mapping is used.

The placement of VMs on the physical hosts works as follows. Each set of 400

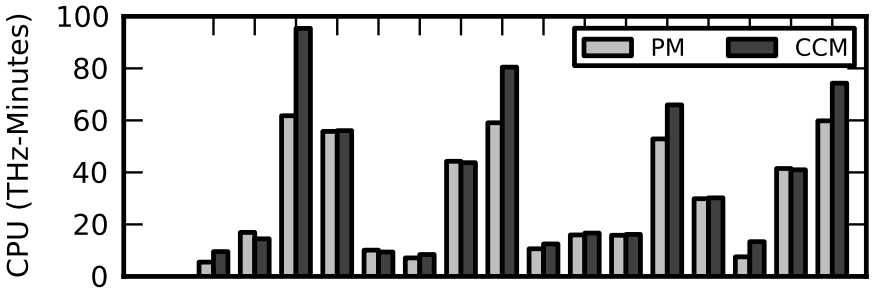
Table 3: Scenario-wise DRS and CCM Parameters. Parameter settings common for all scenrios span the entire entity column.

Param (row-wise)	**Clus/Part**	**S-clus**		**Cloud**	
Scenario (col-wise)	I, II, III	I, III	II	I, III	II
Mon Int. (secs)	20	120		600	
Bal Int. (mins)	5	20		60	
I_{CPU}	$\leq Prio3$	0.15	0.1	0.15	0.1
I_{MEM}	$\leq Prio3$	0.2		0.2	
$Max(Moves_{host})$	n/a	6	8	6	8
$Moves_{host}^{parallel}$	n/a	3	4	4	8
$Timeout_{host}^{move}$ (mins)	3	16		16	
Reac. Int (mins)	n/a	n/a	10	n/a	30

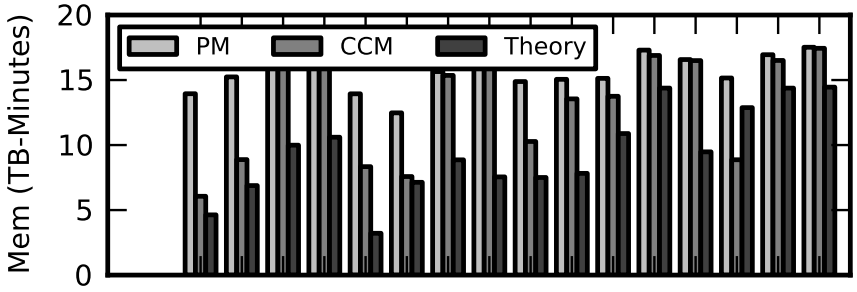
VMs representing a particular job is further sub-divided into sets of 100 VMs and initially distributed evenly amongst the 32 hosts of a cluster or partition. Each VM is configured with 4 virtual CPUs (VCPUs) and 2GB of memory. In the case of PM, VMware DRS dynamically multiplexes the load of the 100VMs on the 32 hosts of each partition using VM migrations. In the case of CCM, each set of 4 clusters are further composed into a single supercluster, and the 4 superclusters in turn report to a single cloud manager, with capacity multiplexing being performed at each of those levels.

The first experiment evaluates CCM's ability to continuously right-size the allocation for each job based on its actual runtime demand. Jobs become under-provisioned when the VMs' total resource demand during any time period is more than the available capacity in their respective cluster or partition, and similarly, become over-provisioned when the total demand is less than capacity. Table 3 shows the configurable parameter settings for DRS and CCM used for this experiment. We've derived these parameters experimentally.

All cluster, job, and cloud utilization numbers reported here and in the other scenarios are aggregates of the individual VM utilizations, and they do not include the resource utilization due to the management operations or virtualization overhead.



(a) CPU

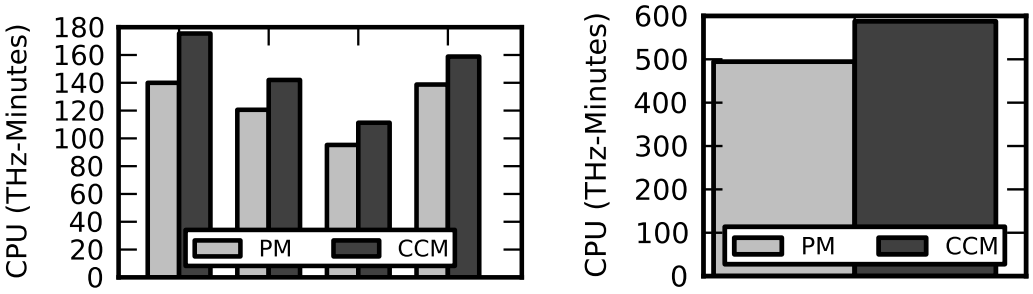


(b) Memory

Figure 12: Cluster-wise/Partition-wise Resource Utilization.

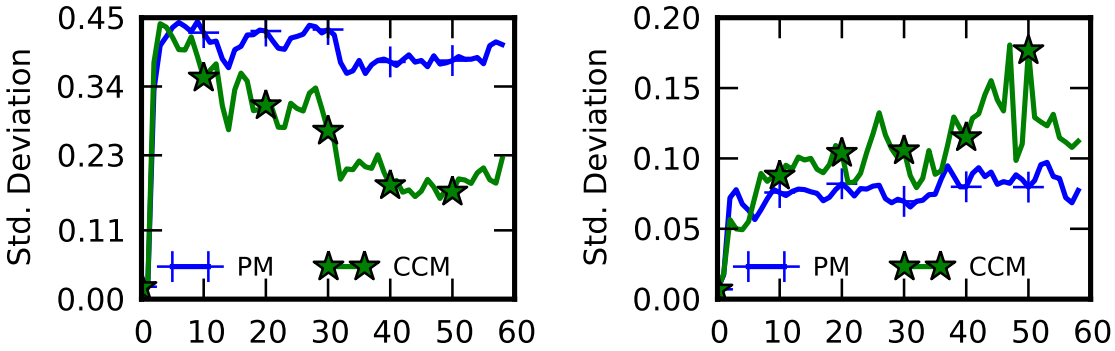
As a result, any improvement in utilization is due to the workloads inside VMs being able to do extra work as more resources are made available to them. Figure 12a shows the cluster-wise or partition-wise CPU usage computed as the Riemann sum of the corresponding aggregated VM usage curve over time, with individual samples obtained once every 5 minutes (average value over interval). Overall, there are a few clusters that require a large amount of CPU resource while the majority require a much smaller amount, over the course of the experiment.

As seen in the figure, statically sizing partitions and managing resources within the partition leads to sub-optimal use of available cloud capacity. In contrast, CCM is able to improve the total CPU utilization of the VMs in high demand clusters by up to 58%, by adding capacity from other unused clusters that are mostly part of the same supercluster. Figure 13a shows the job-wise CPU usage as the sum of their corresponding VM CPU usages. Here again, CCM is able to increase every job's



(a) Jobs (b) Cloud

Figure 13: Aggregate CPU utilization.



(a) Clusters of Supercluster 0 (b) Superclusters

Figure 14: CPU Imbalance in Hierarchy.

utilization, between 14% to 25%, as its multiplexing removes resource consumption hotspots. The additional effect of all of these actions is that CCM is able to improve the overall datacenter VMs' CPU utilization by around 20%, as well (see Figure 13b), compared to a partitioned management approach. Also, as seen in Figure 12b, while CCM has a lower memory utilization than PM for VMs of each cluster, it is still more than what is theoretically required by the workload as computed from the trace. The memory utilization values collected for each VM at the virtualization infrastructure level include both the workload and the guest OS base memory usage.

Figures 14a[2] and 14b provide more information on how CCM operates with respect

[2]Imbalance across clusters is only shown for Supercluster 0; other superclusters follow a similar trend.

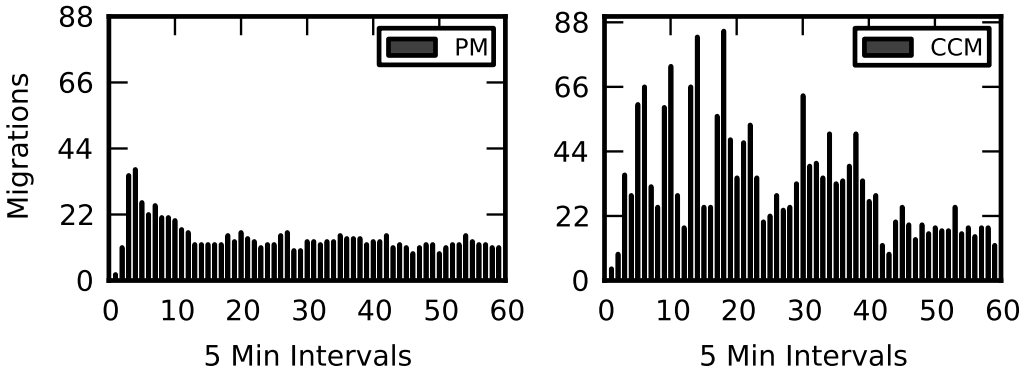


Figure 15: Total VM Migrations every 5 Minutes.

to reducing imbalance. Across the clusters of each supercluster, the CPU imbalance metric starts off high but CCM soon reduces it to the configured threshold of 0.15. Clusters are dynamically grown and shrunk in terms of capacity backing the workload VMs in such a way that their normalized utilization (NE) values are similar. Interestingly, in Figure 14b, CPU imbalance between superclusters for CCM grows higher than for the PM approach. This expected behavior is due to the fact that CCM does not take any action to reduce imbalance as long as it is within the configured threshold of 0.15. In addition to varying workload demands, another factor affecting how quickly CCM corrects resource usage imbalances in the datacenter, is the aggressiveness of its management actions, controlled via active throttling.

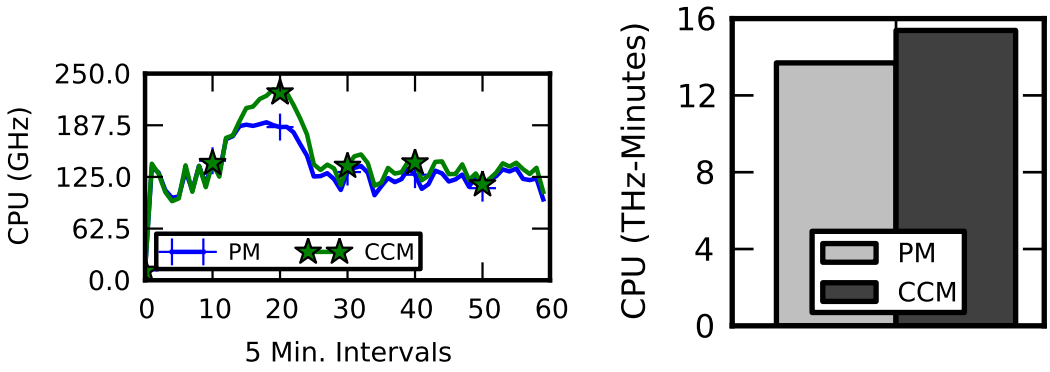
The number of VM migrations performed by PM and CCM across the entire datacenter, every 5 minutes, over the course of the experiment, is shown in Figure 15. Given the increased imbalance inherent in this workload,
CCM's successful balancing of capacity uses twice as many migrations as PM on average – 32 vs. 14, per 5 minute interval, but it is apparent from Figure 13a that the implied management overhead is sufficiently well amortized, ultimately leading to notable improvement in the resources made available to jobs. When considered in the context of the entire cloud, this figure reflects as only 0.9% and 2%, respectively, of all the VMs, being migrated every 5 minutes. One reason for this modest overhead

is that CCM explicitly manages management enforcement, by using timeouts on the overall host-move operation to abort long running migrations, and by promoting the selection of cheaper operations at a later time.

II. Workload Volume Spikes: the previous experiment featured jobs with variable demands but with a gradual rate of change. To test CCM's ability to allocate capacity to workloads experiencing a sudden volume spike in resource usage, as described by Bodik et.al.[39], we use the parameters identified by the authors (duration of spike onset, peak utilization value, duration of peak utilization and duration of return-to-normal) to add two volume spikes to a base resource usage trace. We generate the baseline resource (CPU only) usage trace from enterprise datacenter resource usage statistics reported by Gmach et.al.[56]. We use the statistics about the mean and 95th percentile values of the 139 workloads described in the paper to generate per workload resource usage trace for 5 hours assuming that the usage follows a normal distribution. Groups of 12 VMs in the datacenter replay each single workload's resource usage, letting us cover a total of 1600 VMs on 512 hosts. The cluster and supercluster organizations are similar to the previous scenario. The first volume spike quickly triples the overall workload volume over the space of 20 minutes across a subset of 100 VMs, mapping to a single cluster/partition, during the initial part of the experiment. The second spike triples the overall workload volume somewhat gradually across a larger scale of 400 VMs, mapping to a single supercluster, over the space of an hour, during the latter part of the experiment.

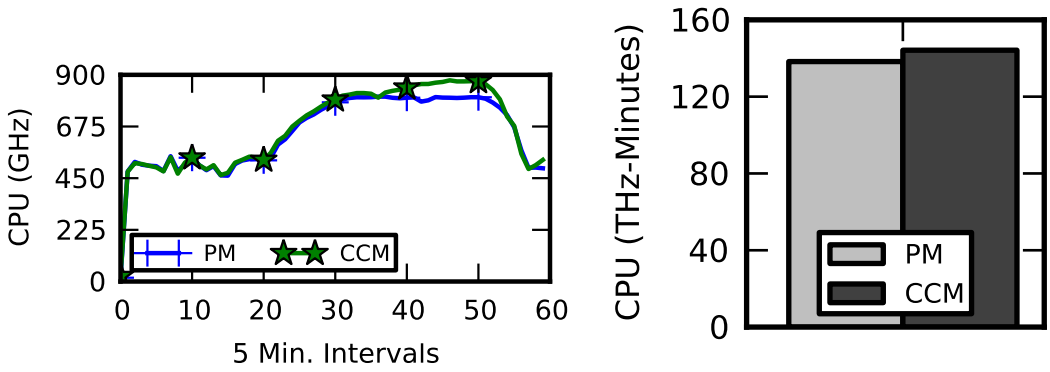
Table 3 shows the configurable parameter settings for this scenario. The central free host pool is set to hold DPM recommended hosts of up to 10% of the total supercluster capacity. The appropriate hosts are harvested from the clusters of each supercluster at the end of every periodic balancing round. Note also that in this scenario the reactive balancing operation does not have any limits to the number of hosts that can be moved from the central free host pool to a resource starved cluster

or supercluster. This is due to the fact that these hosts are already pre-evacuated, resulting in the corresponding management action having lower complexity and cost. The overall behavior of the system in this scenario would be such that the majority of capacity movements happen due to reactive balancing, with periodic balancing only correcting any major imbalances in sub-entity utilizations.



(a) Util. over Time (b) Util. Area of Spike

Figure 16: Cluster Spike: Aggregate CPU utilization. Spike Duration = Samples 10 to 25 (75 mins).



(a) Util. over Time (b) Util. Area of Spike

Figure 17: Supercluster Spike: Aggregate CPU utilization. Spike Duration = Samples 20 to 56 (180 mins).

Figure 16 shows the total VM CPU utilization achieved, for both PM and CCM, in the cluster replaying the sharper spike across 100 VMs. CCM is able to achieve a 26% higher peak value compared to PM (see Figure 16a). Overall, this translates to a 15%

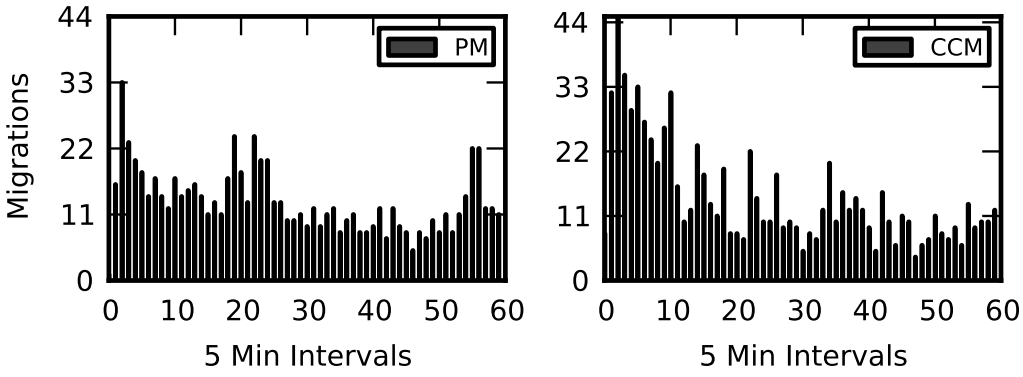


Figure 18: Total VM Migrations every 5 Minutes.

improvement in total CPU utilization over the duration of the spike (see Figure 16b). With the larger magnitude supercluster level spike though, CCM's improvements are a modest 13% for the peak CPU utilization value (see Figure 17a) and 4% for the overall CPU utilization over the duration of the spike (see Figure 17b). Given the larger scale of the spike, a higher improvement would require the quick movement of a correspondingly larger number of hosts.

Our ability to accomplish this is constrained by the capacity of the central free host pool at each supercluster and also the rate at which the free host pool can be replenished with hosts. In addition, the host harvesting operation at the end of each periodic balancing round still has a limitation on the number of hosts that can be moved in order to reduce move failures and management cost. Depending on the importance of dealing with, and, the relative incidence of such aggressive volume spikes, the datacenter operator can choose to tailor the configurable parameters to trade between the potential loss of performance due to holding too many hosts in "hot-standby" vs. the improved ability to deal with volume spikes.

In terms of total migration costs, as seen in Figure 18, CCM has only a slightly higher migration rate compared to PM (14 vs. 13 on average, per 5 minute interval). The extra migrations are mostly due to the host harvest operations that happen at the superclusters. The DPM module is fairly conservative in choosing candidate hosts to

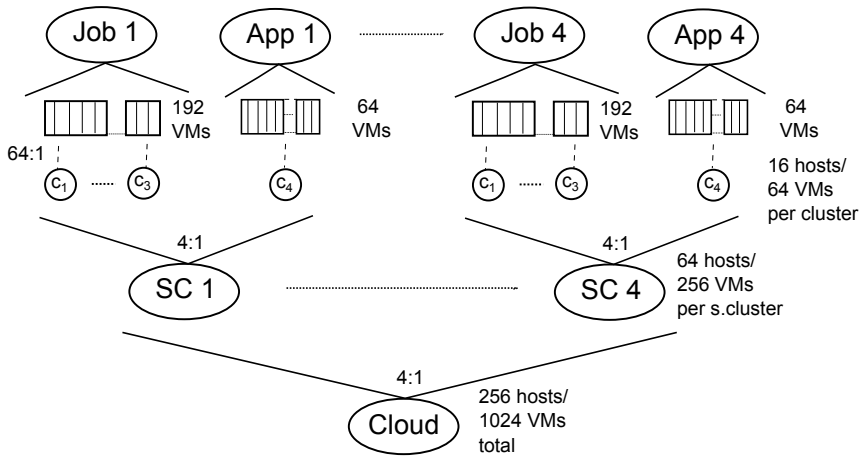


Figure 19: Application Performance Scenario Architecture.

be powered down; sometimes even choosing none. This reduces the number of hosts harvested during each invocation and the number of migrations being performed.

III. Application Performance: CCM's multiplexing should improve the performance of applications experiencing high usage or increased resource constraints due to co-location, without significantly degrading the remaining workloads. To evaluate this capability, we construct a scenario in which the cloud runs a workload consisting of the 4 jobs from Scenario I and an instance each of Nutch, Olio, Linpack and Voldemort. This arrangement allows us to capture each application's behavior in a datacenter environment with a realistic background load pattern. We run the workloads using 1024 VMs running on 256 servers in the datacenter. Almost all the VMs have equal Shares with no Reservation and Limit control values except for the Nutch, Linpack and Voldemort master nodes that have a memory Reservation of 1GB each given their relative importance. The overall physical and workload VM organization is shown in Figure 19.

Table 4 shows the raw average application performance metrics and the coefficient of variation (CV) of each metric, across 4 runs each of PM and CCM. The last column shows the percentage change in performance using CCM compared to PM. It is apparent that by dynamically allocating capacity to back the actual resource

Table 4: Application Performance Metrics.

App.	Metric	PM		CCM		% diff
		Avg	*CV*	*Avg*	*CV*	
Nutch	Jobtime (mins)	134	0.55	106	0.28	21
Linpack	MFLOPS	392	0.04	468	0.11	20
Voldemort	Ops/s	10.81	0.42	9.8	0.5	-9
Olio	Reqs/s	33	0.65	274	0.6	730

demand, CCM leads to noticeable performance improvements for Nutch, Olio and Linpack with a mild performance penalty for Voldemort. The Olio application is both CPU and memory intensive during its operation. The reason for the extremely high improvement seen with Olio is due to the fact that, unlike high CPU pressure, an extremely high memory pressure causes the Olio VM guest operating system to invoke its local memory management functions such as swapping and OOM killer, during the course of the experiment. As a result, request throughput suffers more than linearly, as in the case of PM, when such memory pressure is not alleviated with additional capacity.

An important factor to point out in these results is the non-trivial variability in performance observed for the applications (CV values in Table 4). Some of this variability is due to the fact that management operations in both PM and CCM (i.e., VM migrations and host-moves) have different success and failure rates across multiple runs. This leads to a different resource availability footprint for different runs causing variability in performance. In addition, the current version of CCM also has limitations due to the fact that it does not explicitly take into account datacenter network hierarchy, rack boundaries etc. in its VM migration decisions. We believe that upcoming fully provisioned datacenter networking technologies [57, 83] will obviate some of these concerns.

3.5 Related Work

Broadly, resource management solutions for cloud datacenters can be classified as application-driven, infrastructure-driven, or a hybrid of both. Among the infrastructure level solutions, Meng et.al. present a system to identify collections of VMs with complimentary resource demand patterns and allocate resources for the collection so as to take advantage of statistical multiplexing opportunities [79]. Such opportunities are inherent in cloud datacenter VM collections given the many customers and workloads they host. Wood et.al. present a system that uses black-box and gray-box information from individual VMs to detect and alleviate resource hotspots using VM migrations [96]. Chen et.al. provide and O(1) approximation algorithm to consolidate VMs with specific demand patterns onto a minimal set of hosts [43]. These solution approaches are designed to inspect individual resource signatures to allocate and extract good multiplexing gains in the datacenter.

There also exist systems that use explicit application feedback to tune resource allocation[73, 85, 99, 91, 82] at fine granularity (e.g. MHz for CPU) and coarse granularity (e.g. add/remove servers) in order to maintain SLAs. Such systems can be built on top of low-level capacity management systems like CCM to better achieve the objectives of both the cloud provider and customers. CloudScale [90] is an elastic resource allocation system that monitors both application and system parameters, primarily focusing on the accuracy of allocation methods in order to balance workload consolidation with SLA achievement. As the authors point out, this system is designed for a single datacenter node and can complement a large scale system like CCM. Zhu et.al. construct a resource allocation solution that integrates controllers that operate at varying scopes and time scales to ensure application SLA conformance [100]. CCM shares a similar architecture. The differences lie in the fact that CCM's mechanisms are designed for scalability (e.g. moving capacity vs. VMs), tackling management operation failures and cost variability. Further, we envision

CCM as a low level datacenter capacity management system that exports interfaces through which application level solutions interact to maintain SLA conformance.

It is rather well known that large scale management systems must be designed to work in the presence of machine and component failures[77, 65]. However, the presence and the need to design for failure of *management operations* has received much less attention, if any. In addition, management operations also have bursty and highly variable resource consumption patterns [93, 94] that if not kept in check, may lead to unacceptable overheads that degrade application performance. CCM shares the design philosophy of using conservative but reasonably effective methods in this regard, over complex ones, with other large scale datacenter systems such as Ganglia [77] and Autopilot [65].

3.6 Chapter Summary

This chapter describes a composite set of low-overhead management methods for managing cloud infrastructure capacity. Most real life datacenters have varying hardware and software limitations that prevent aggressively carrying out management operations. We have demonstrated, through data from an experimental evaluation on a fairly large infrastructure, that to achieve better capacity multiplexing, the focus needs to not only be on the accurate prediction of workload demand and aggressive optimization of the allocation algorithms, but also on dealing with the practical limitations of real-life infrastructures. While in this work, we stress the need for, and, use *simple* methods to overcome the detrimental effects and achieve good performance, our next system, Speculor, uses design guidelines obtained from an analysis of the data on these overheads and failures, and, develops a systematic approach to tackling them.

CHAPTER IV

AN ANALYSIS OF VM LIVE MIGRATIONS IN THE WILD

Our last chapter showed that in large scale cloud computing environments: (1) management operations may fail at non-trivial rates, and, (2) there can be large variations in the costs of such management operations. Failures decrease the effectiveness of capacity multiplexing, and variable costs complicate dealing with management overhead relative to the benefits being derived. In this chapter we aim to further quantify their behavior and analyze the underlying causes using data obtained from the evaluation of CCM and its early prototypes. This chapter sheds light on why CCM performs well and also motivates the need for more targeted fault-tolerance approaches, such as the one discussed in the next chapter. We distill a set observations that will also be useful for any system designers using virtual machine live migrations for building automation services in datacenters such as high availability, disaster recovery, rolling upgrades, power management, cloud application deployment and orchestration etc.

4.1 Anatomy of a Host-move Operation

Recollect that a *host-move* is one of the basic actions performed by CCM at the supercluster and cloud levels, the purpose being to elastically allocate resources in the datacenter. There are 2 major kind of moves: (i) host-move between clusters, and (ii) host-move between superclusters. Each such action is composed of a series of 'macro' operations in the management plane that must be executed 'in order', as shown in Figures 20 and 21, respectively. Each such macro operation may be implemented by one or more lower level 'micro' management plane operations.

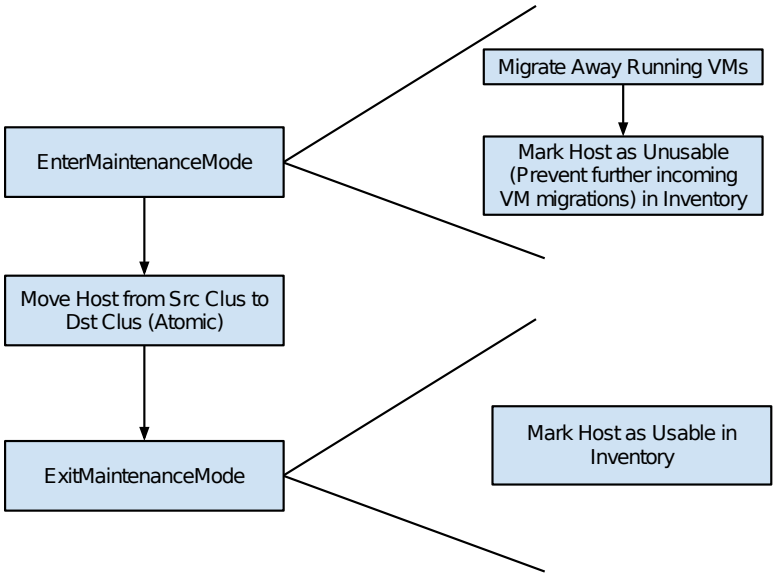


Figure 20: Inter Cluster Host Move

The "EnterMaintenanceMode" operation, for instance, places a host into a "maintenance" state in which no VMs currently use it, so that this host can then be moved from one cluster to another. This is potentially the most resource intensive of these management plane operations. Its resource intensity is due to the need to evict from the host all VMs currently running on it, where resource intensity and thus, the duration of the operation is governed by factors that include VM size and active VM memory footprints [45]. Evicted VMs are moved to other hosts in the source cluster selected using DRS. In the case of superclusters, they typically encompass a large number of hosts that span multiple disjoint vSphere Virtual Center (VC) instances. Hence, an inter-supercluster host-move operation may also include the removal of a host from the source supercluster VC inventory and addition of the same to the destination supercluster VC inventory.

The important point to note about these composite host-move operations is that the failure of a macro operation always results in complete failure of the whole host-move, whereas a failure of a micro-operation may or may not result in total failure

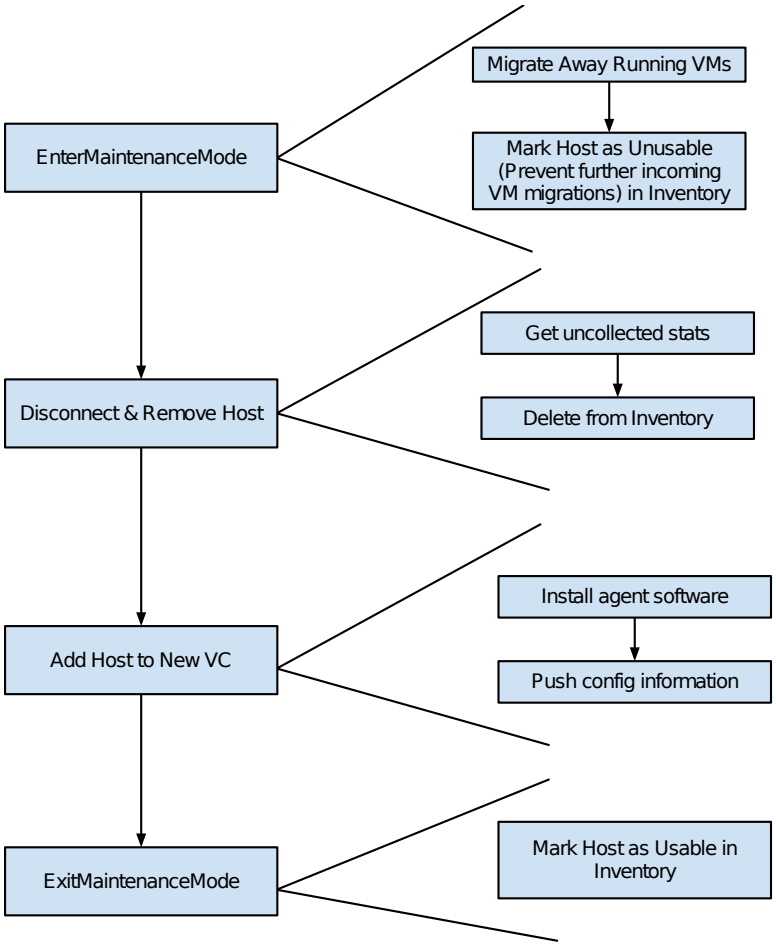


Figure 21: Inter Supercluster Host Move

depending on whether it is a pre-requisite for future operations (e.g., "Getting uncollected stats" need not result in total failure). This classification of management plane operations is useful when devising ways to cope with failures.

Figure 22 shows the proportion of successful host-move operations out of those attempted overall for all of the four different CCM configurations running the experiment from Scenario I in the evaluation section of Chapter 3 on CCM. All of the host-moves analyzed here are inter-cluster moves. We observe a 38% host-move failure rate and low number of successful host-moves, even in the best case for CCM with timeouts and thresholding. In addition to outright failures, we also observe a large number of extremely slow operations that did not complete during the course of the experiment. In the figure, these are also counted as failures.

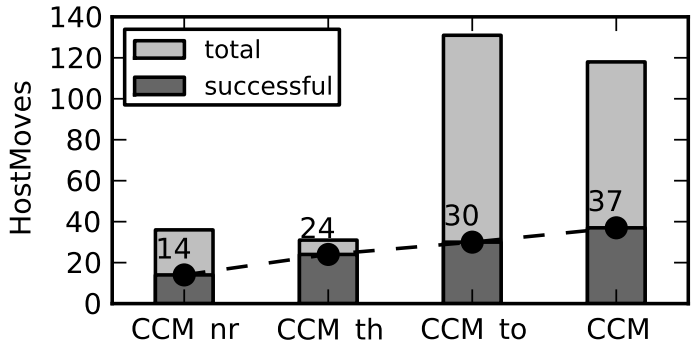


Figure 22: Number of successful host-moves

In the case of CCM_to and CCM, both configurations achieve a much higher success rate while also attempting almost 3 times as many host moves as CCM_nr and CCM_th. This is because having a timeout allows quickly backing out long running host-moves whose cost to workload benefit ratio is likely highly unfavorable. If the load imbalance in the workload continues to persist, the balancing algorithms recommend a fresh set of moves during the next round that may have a higher likelihood of being cheaper.

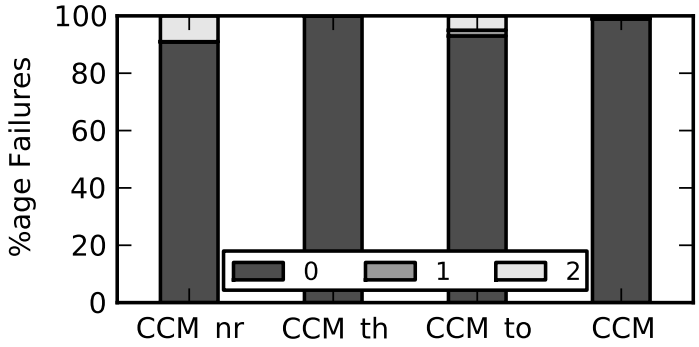


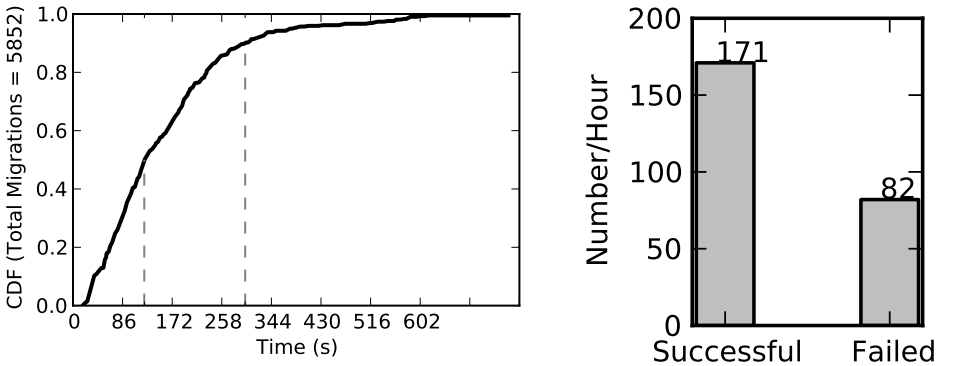
Figure 23: Fraction of inter-cluster host move failures due failure of each macro operation. Key: 0 - EnterMaintenanceMode, 1 - MoveHost, 2 - ExitMaintenanceMode.

In order to gain a deeper understanding of the causes of these observed behaviors, Figure 23 presents the proportion of failures due to a failure of each of the 3 macro management plane operations in the inter-cluster move. It can be seen that more than 90% of the failures are due to a failure of the "EnterMaintenanceMode" operation, or, in other words a failure to evict (by migrating them away) all of the running VMs

on the hosts in question.

Therefore, the failure and non-deterministic management operation behavior observed in CCM are attributable to the behavior of VM live migrations. We next analyze the characteristics of VM live migration using data obtained from our environment in an attempt to identify root causes that explain their failures.

4.2 Quantifying VM live migration behavior



(a) CDF of VM migration times (b) Avg. No. of Migrations/Hour

Figure 24: VM Migration Statistics

Figure 24a shows the cumulative distribution function of all the duration of all VM live migrations performed as part of evaluating CCM in Scenario III. This dataset contains a month's worth of VM migrations on 256 servers in our datacenter. The workloads consisted of both trace replay and the four applications, as discussed in the previous chapter. The data shows that the VM live migration times have a long tail with the 50%ile latency being 110s and the 90%ile latency at 287s or almost 5 minutes. Prior studies have shown that the total resource cost, i.e., network bandwidth usage, of a migration is directly proportional to its duration [33]. In the popular iterative pre-copy based VM migration implementation, the longer a given pre-copy round lasts, the larger the amount of data that will be dirtied by the migrating VM.

Table 5: VM migration failure causes breakdown for each configuration. Values denote percentages.

Abbreviated Cause	**CCM_nr**	**CCM_th**	**CCM_to**	**CCM**
General system error	31	22	10	21
Failed to create journal file	45	7	63	54
Operation timed out	4	49	18	13
Operation not allowed in cur state	12	22	4	12
Insuf. host resources for VM reserv.	0	0	3	0
Changing mem greater than net BW	0	0	1	0
Data copy failed: already disconnected	8	0	0	1
Error comm. w/ dest host	0	0	1	0

Therefore, this graph indicates a large magnitude of variation of the cost of each individual VM migration (roughly two orders in our infrastructure) in real-life datacenter environments running representative applications. Solutions that assume uniform or negligible migration costs are unlikely to be effective in practice.

Figure 24b shows that on an average roughly 32% of all VM migrations attempted during our experiments failed in one way or another. Table 5 shows some of the major causes of VM migration failures collected from the vSphere layer and their percentage contribution to the overall number of migration failures. Firstly, it can be deduced that there are a large number of reasons why VM migration operations fail in the wild. The administrator visible error messages, though useful, typically relay failures as perceived by higher layers of the virtualized management stack. The true causes for observed failures may actually be further varied such as software timeouts in different layers, misconfiguration, bugs, network connectivity issues etc. as observed in other similar complex software environments [65, 62].

Secondly, some of the observed causes from Table 5 point to the fact that migrations may have failed directly or indirectly due to network bandwidth insufficiency (e.g. "Operation timed out."). We develop this point further by statistically analyzing the relationship between a VM migration outcome and the number of other simultaneous ongoing VM migrations competing for available network bandwidth.

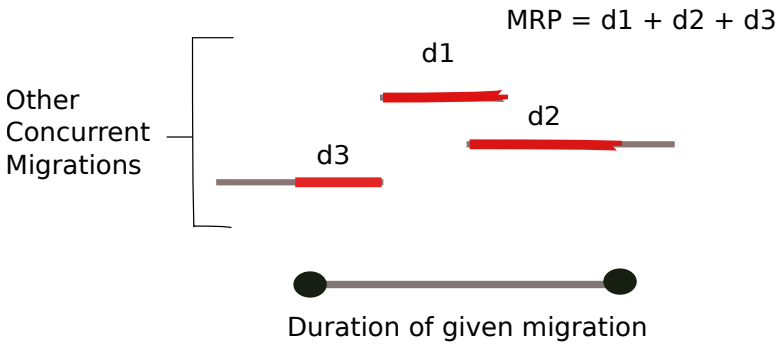


Figure 25: Management Resource Pressure (MRP)

Recall, that in our datacenter we use a flat network topology consisting of a single high port-count switch. We capture this relationship by defining a new metric called *management resource pressure* - it is the amount of temporal overlap between a given migration's duration and other concurrent migrations competing for network resources. We illustrate this metric pictorially in Figure 25. As described previously, time duration is a good proxy for resource consumption of a VM migration. Intuitively, the higher the temporal overlap, the higher the disruption for the given VM migration.

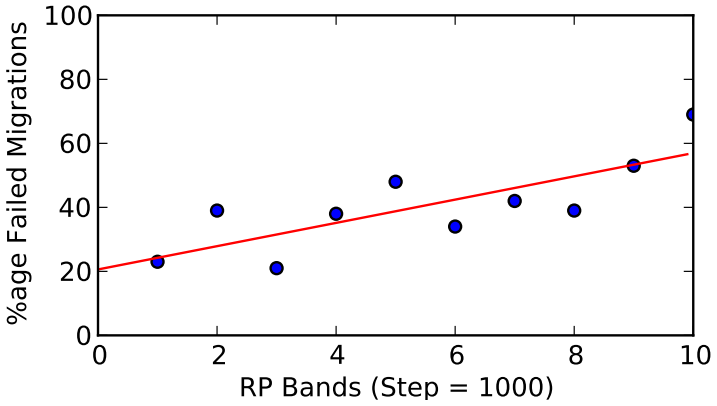


Figure 26: Management Resource Pressure (MRP) vs. Migration outcomes

Figure 26 plots fraction of failed VM migrations for each band of thousand MRP units. Note that the exact value of MRP does not matter as much as the fact that adjacent bands differ noticeably in their network contention. In other words, MRP

bands capture the amount of network contention during a given VM migration. We observe a Pearson's correlation coefficient of 0.8 between the two factors indicating a high positive relationship validating our previous intuition. However, the coefficient of determination for the fit indicates that only roughly 60% of the failures can be attributed to network insufficiency.

In summary VM migration failures happen due to large number of reasons with network bandwidth insufficiency featuring prominently in the list of probable causes for outright failure or those operations that exhibit extremely poor performance.

4.3 Chapter Summary

In this chapter we've established (1) the magnitude of variation in VM migration resource costs by analyzing their duration and (2) the significant rate of VM migration failure in large scale environments due to a wide variety of reasons and the key role network resource insufficiency plays in inducing failures. Some of these failures may manifest themselves as poorly performing migrations that consume a lot of resources before ultimately failing. CCM uses simple timeouts to backout long running operations and try them again during the next invocation of its algorithm - maintaining good cost vs. benefit of management using a simple design. Together with host-move, and transitively VM migration, thresholding prevents potentially failing migrations from wasting too much network resources.

However, there is a clear need to design *fault-agnostic* fault-tolerance mechanisms for capacity multiplexing solutions given the wide variety of causes underlying VM migration failures. The system described next tackles this problem while being cognizant of the limitations imposed due to available infrastructure network capacity.

CHAPTER V

SPECULOR: FAULT-SCALABLE CAPACITY MULTIPLEXING

5.1 Introduction

As seen from the data presented thus far, in large scale virtualized datacenters, given the scale and complexity of the deployed software at various layers, and the ubiquity of commodity hardware in these environments, compute capacity multiplexing systems are routinely faced with a large number of failures of VM migrations. On a general note, such failure behavior has also been reported of other operations on VMs like power-on, snapshot etc. [76], and on other virtualized entities like virtual networks [89, 97], likely affecting most infrastructure automation services in the virtualized datacenter.

The state of the art in dealing with such commonplace failures can be categorized into two schools of thought: (a) offer little to no support at the infrastructure management level and rely on end application-level graceful fault-tolerance methods, (b) employ root-cause diagnosis approaches developed for complex distributed systems. Most commercially available infrastructure management software (e.g. VMware vCloud [23], OpenStack [17]) that enable private cloud deployments and public facing clouds (e.g. Amazon EC2, HP Public Cloud) fall under the first category.

Such an approach exposes customers to a large majority of infrastructure level failures and burdens application developers with developing reliability mechanisms alongside the already complex business logic. The situation is made worse as no infrastructure level information is typically provided to them, preventing them from intelligently employing resilience mechanisms as needed to balance costs vs. benefits.

The second category, root-cause diagnosis systems, are useful in identifying and fixing failure causes in the long run. They often require manual intervention to confirm and eventually fix and deploy patches resulting in a high turnaround time. This makes it infeasible to solely rely on root-cause diagnosis systems on an ongoing basis. Further, the large variety of factors underlying failures in the datacenter environment, makes pinpointing them a difficult endeavor.

To address these issues, we describe the design and implementation of a fault-scalable speculative virtualized infrastructure management stack. We achieve this by *speculatively executing multiple logically equivalent replicas of basic virtual machine operations like live migration simultaneously across different physical targets* like servers, switches and network links with independently failing hardware and software components.

The virtualized infrastructure management stack exposes a speculative version VM live migration while semantically ensuring the following correctness properties: (i) only one speculative operation commits and persists out of all the replicas, (ii) the intermediate state of a VM during speculation is not exposed to layers above where the speculative operation is implemented. The policy aspects of employing speculatively replicated live migrations (i.e., choice of physical targets, degree of speculation etc.) are left up to higher layer automation services like capacity multiplexing. The stack also supplies automation services with relevant information on observed failure behavior and potential costs, through a querying interface, so that they can better manage the cost vs. benefit of speculation. Such a design leads to a clean separation of concerns.

Our speculative extensions and capacity multiplexing for this environment have been built into the popular open source OpenStack infrastructure virtualization platform running Xen hypervisor based servers. We demonstrate the usefulness of our

concepts using simulations and direct experiments on a 12 node cluster for parameterized failure scenarios.

5.2 Fault Model

Studies on observed service disruption events in large scale datacenters have repeatedly pointed out that roughly 60-80% of all causes are due to software issues (bugs, misconfiguration etc.) and human mistakes [84, 62, 38, 66, 54]. This stands as a testament to the significant improvements that have been made over several decades in developing software for tolerating fail-stop hardware faults which are straightforward to detect via liveness checking mechanisms like heartbeats and timeouts. Software faults on the other hand, especially in production software that has undergone several cycles of testing, tend to be *intermittent and transient* in nature making them harder to detect or predict. Operator mistakes have also long been known to create *correlated failure* scenarios affecting several systems at the same time.

To complicate things further, with ever shortening software release cycles [9, 21] and the use of independently developed third-party software components, *newer and evolving failure modes* are introduced at a faster pace, making it infeasible to solely rely on techniques like root-cause diagnosis systems [35, 53, 98, 42]. It is simply untenable to get to the bottom of every possible failure in a datacenter software.

In addition, our discussion of resource insufficiency in the previous chapter and recent work on performance degraded hardware [49] point to the fact that failures also manifest themselves as *poorly performing operations* that consume arbitrary time and resources. These failures have the potential to affect other co-hosted applications and customers in the datacenter.

Therefore, we aim to provide mechanisms that provide *fault-scalable* performance of management operations, specifically for the above highlighted scenarios. It is imperative that our approach is as *fault-agnostic* as possible. We also assume that the

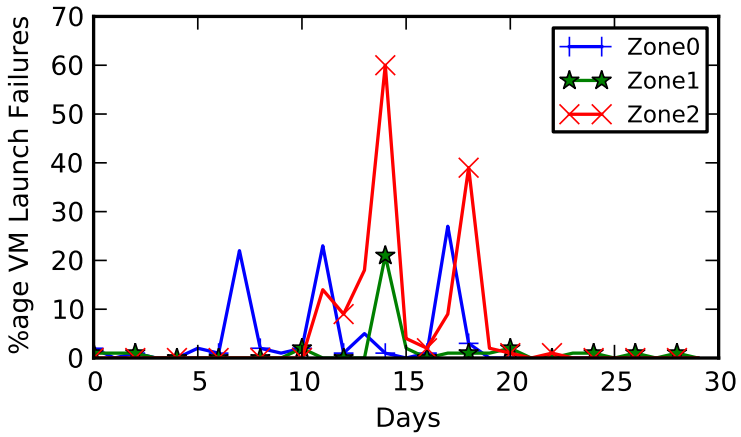


Figure 27: Daily percentage of VM instance launch failures for the month of November 2013 in three zones of HP Public Cloud.

faults are non-byzantine in nature, which is fair in the datacenter environment with a high degree of administrative control. Further, we want to build fault-tolerance at the lower-level virtualization and automation layers of the datacenter software stack in order to be customer *application transparent*.

Unlike web service-level and hardware failures, very few studies exist that report statistics on the failure of operations on virtualized entities like VMs in the wild. Mao et.al. report their experience benchmarking VM instance launch operations on Amazon EC2, Microsoft Azure and Rackspace Cloud observing that on average launch failures are in the range of 0.4% to 8% on these clouds over the entire course of their experiments [76]. This is consistent with the observations made by Ravello Systems Inc., a large consumer of public cloud computing resources, which reports observing between 1% to 10% average launch failures over a two week period [5].

These seemingly low failure fractions belie the true burstiness of operations on VMs and their failures due to their summarization over long periods. Figure 27 plots the VM instance launch failure fraction observed by Ravello Systems Inc. at HP Public Cloud in November 2013, on a per-day basis, using data made publicly available [4]. While the overall monthly failure fractions are low it can be seen that

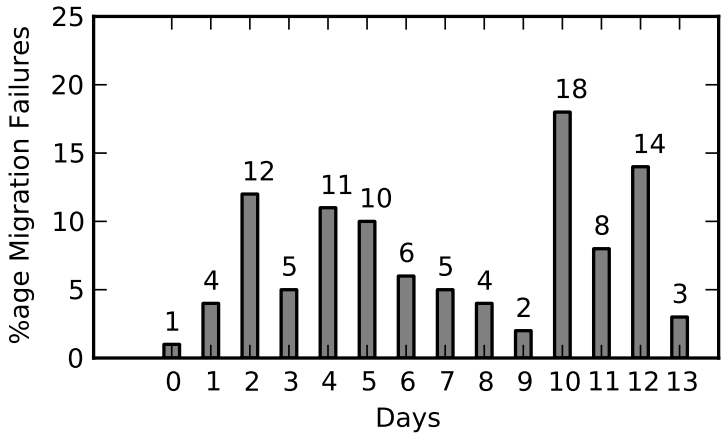


Figure 28: Daily percentage of VM migration failures over a two week period in September 2012 at a 700 server research cluster at Georgia Tech.

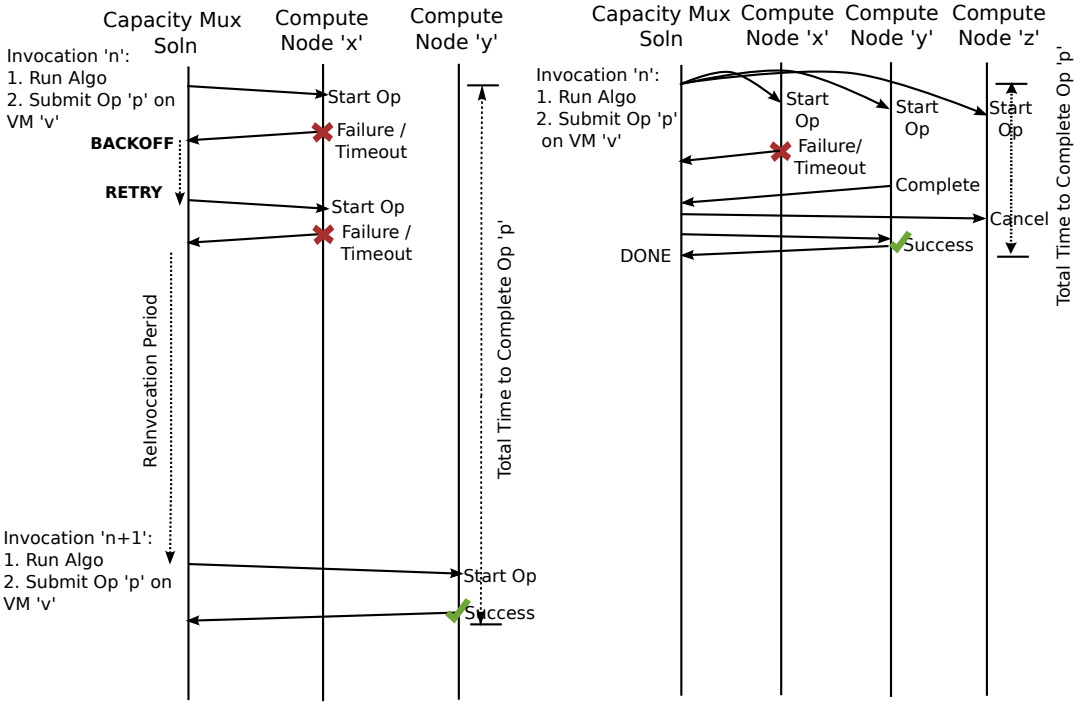
on certain days failures can be as high as 61% of all VM launch operations attempted.

The burstiness of virtualized datacenter management operations is also documented by Soundararajan et.al. in their study of data collected from several enterprise datacenters running the VMware virtualization platform. They report several orders of magnitude difference (upto 300x in one case) between the average and peak number of operations executed on VMs. In our own 700 server research cluster at Georgia Tech running VMware vSphere 4.1 virtualization platform, we've observed a large variation in the daily VM migration error percentages over the course of two weeks as shown in Figure 28.

The above presented evidence underscores the dire need for techniques such as the one we propose in this chapter. It also pinpoints the importance of judiciously using speculative replication of operations, especially for resource-intensive ones like VM live migrations, to optimize the cost vs. benefit of tolerating failures given the wide variability in observed failure fractions over time. Our design explained next attempts to satisfy all the criteria listed in this section.

5.3 Design

5.3.1 Approach



(a) Replication in Time (b) Speculative Replication in Space

Figure 29: Replicating actions in space provides better fault tolerance and performance.

Recall that our primary design goals are to achieve fault-agnosticism and fault-scalable performance. Data replication has been widely used in modern distributed systems to improve data availability and performance [47, 55, 46] - the precise two properties we seek in the context of operations on virtual machines. Our approach, therefore, uses replicating operations (i.e., function replication) to achieve our objectives. However, there are several differences in the semantics and realization of replicating operations compared to data replication as explained in the next subsection.

Consider the example of migrating a VM to a different host in order to allocate more resources to it - a task common to capacity multiplexing solutions. Figure 29

illustrates how our replication model improves fault-tolerance and performance compared to techniques available in current generation systems. Most infrastructure virtualization solutions like VMware vCloud and OpenStack employ retrying failed VM migrations for a fixed number of times with optional backoff between retries [8, 28] as shown in Figure 29a. Intermittent failure of operations and nodes are common and inevitable in large scale datacenters due to a large variety of reasons. This mode enables some tolerance to such events.

Retrying involves executing the same operation after failure is asserted with the same parameters i.e., VM, source and destination hosts, but at a different time. Given that the duration of the failure event is unknown, retrying may end up wasting resources in multiple attempts and also elongating the eventual completion time of the original intended migration operation, as shown in the Figure 29a. During this time, the VM is unable to get the amount of resources it requires to run its application.

When all retry attempts fail, the system falls back to the live migration operation on the VM being recommended again by the capacity multiplexing algorithm in its next invocation, if it still requires more resources, this time with a potentially different host, as it may deem fit based on the changed global state in the intervening time. When this happens, the delay in completing the originally intended VM migration is the highest, leading to poor VM performance.

In the preceding discussion it can be seen that there emerges a *notion of a logical operation on a VM as perceived by the capacity multiplexing service* - migrate a VM to allocate more resources to it. This logical operation can have multiple physical manifestations on different servers across two subsequent invocations of multiplexing algorithm. This property is the basis for our version of replicating basic operations like live migrations on VMs. It is important to note that when datacenter failure events are uncorrelated, the likelihood of two or more servers experiencing a failure during a the same given interval of time is vastly reduced. Therefore, we replicate operations

in space simultaneously, taking advantage of this property and the fact that there is a natural choice for executing "logical operations" on VMs on different physical targets. Since the operation replication is simultaneous and operating without certain knowledge of the failure behavior, they are executed speculatively with the correctness properties previously alluded to.

Figure 29b illustrates our proposed operation replication in space. When the multiplexing algorithm decides to migrate a VM away from its current server to allocate more resources to it, it picks more than one potential destination host at the same time. For example, the best and the next 'n' best hosts in the ranked order, where the ranking in this case could be based on the amount of free resources available on each host. It then simultaneously migrates the VM to multiple destination servers, using the help of our speculative virtualized infrastructure management stack, during which some may fail while at least one may succeed with a high likelihood. The first replica to complete the operation is chosen as the successful replica and others are aborted. In addition to tolerating failures, replication in space can also take advantage of fast servers or network paths compared to trying an operation singularly. This results in an overall reduction in the time required to complete an operation leading to better resource allocation outcomes.

However, for resource intensive operations like VM live migrations this may incur high overheads in trying to achieve better fault-tolerance. We describe three techniques in the subsequent sections to give the capacity multiplexing service complete control over the cost vs. benefit tradeoff.

5.3.2 Speculative VM Migration

5.3.2.1 Interface and Semantics

Speculative VM migrations are implemented as part of the virtualization layer of the infrastructure management stack. The speculative VM migration operation is exposed via the interface shown in Table 6. The call takes a single input VM and

Table 6: Speculative Migration Interface

spec_migrate_vm(IN vm, IN dsthosts[], IN timeout, IN progress_rate_abort)

instead of a single destination host as in the regular case, a set of unique destination hosts to which the VM will be simultaneously migrated to, speculatively. There is also a timeout, specified in seconds, that sets an overall bound on the composite operation. An alternative version of this API could also include an abort of the composite operation based on total resource usage, say, the amount of network data sent, for example. Our implementation discussed next will require minimal changes to support that.

The last parameter (value in percentage between 0-100) allows terminating all but one of the speculative replicas at any point during the speculative migration, based on the progress of the overall operation. For example, a value of 30 would terminate all but one of the replicas when at least one of the replicas has progressed 30% to completion. This replica is also chosen as the sole survivor of the speculation. This option allows for early termination of replication, bounding its cost. The notion of progress rate is discussed in more detail later.

This mode of co-operatively managing the trade-off between fault-tolerance and its cost is a key feature of our system and is far superior in capturing the requirements of all parties involved than those solutions where all the logic is implemented either in infrastructure management stack or at the application layer. It also leads to a design with a clean separation of concerns.

The infrastructure management stack implements the speculative migration operation and offers the following correctness properties:

- The intermediate state of the VM during speculative replication is not visible externally at layers above the virtualization layer, where it is implemented.

- Only one among all simultaneous replicas will eventually commit its VM state.

These two properties ensure that as a result of executing a VM migration to different physical servers, at no point during or after the operation, does the VM appear as present in more than one place to other VMs or infrastructure services. They essentially provide transparent replication and the VM's state is always consistent to layers above the virtualization layer. The VM migration operation implemented in most hypervisors treat the operation as a transaction with multiple stages that either commit as a whole or abort. We extend this notion to speculative migrations as will be explained in detail shortly.

However, it is inevitable that the state of the physical servers to which a VM is being speculatively migrated to, cannot be made transparent. For example, during a speculative migration, all the destination servers involved need to have a chunk of resources (CPU, memory etc.) reserved for the migrating VM. This reservation on a given host may eventually be not needed if the speculation fails or is aborted in favor of another host. On the other hand if the given host is indeed chosen as the winner among all the speculative replicas, hiding the resource reservation to another concurrent operation like VM power-on or migration, will result in having to fail either one of the concurrent operations when there is not enough capacity to commit both. To avoid potential failures due to such a situation, we choose to expose the capacity state of a physical host during speculation to higher layers, which may then better use that information to pick a service-specific optimal alternative as required.

5.3.2.2 Implementation

We implement speculatively replicated migrations as part of the Xen [37] hypervisor and expose the functionality via OpenStack's nova compute API. A detailed discussion on the original implementation of Xen's VM live migration appears in [45]. Most other hypervisors use similar methods for their migration implementation; so

the following discussion is widely applicable.

To recapitulate, VM live migration design involves moving the execution context of a VM i.e., its CPU, memory and device states, to another host. Memory state of a live VM is typically transferred iteratively in multiple rounds, as pages are being modified, followed by a short stop of the VM and copy of the final set of dirty pages and CPU context. This method is called as iterative pre-copy migration. VM storage is typically assumed to be available on a network-attached storage (NAS) device obviating the need to move entire virtual disks as part of a live migration. The network identity of the VM is typically transferred in the last step of the migration process via an unsolicited ARP reply from the source host. The whole migration process is viewed as a transaction consisting of the following six stages: Stage 1 - Resource reservation, Stage 2 - Base memory image snapshot transfer, Stage 3 - Multiple rounds of iterative pre-copy, Stage 4 - Stop and copy, Stage 5 - Commitment and Stage 6 - Activation.

When a speculatively replicated migration operation is launched, migration to each destination server is handled by a separate thread in the source server. All threads progress through the above mentioned stages in order until either failure or an explicit abort by the migration control software. The bulk of our changes to realize replicated migrations are in the memory transfer code. Specifically, tracking the dirty pages for different migration threads.

The original VM live migration implementation maintains a single dirty bitmap, structured as a radix tree with a single bit tracking each VM page at the leaf level. To log dirty pages, Xen inserts a shadow page table underneath the running guest OS instead of OS's own page tables, as is usual. The shadow page table is populated on demand and all of its page table entries (PTEs) are initially marked as read-only. Whenever the guest OS tries to write to a page, it results in a page-fault to the Xen hypervisor which then consults the guest's original page table to check if write access

is permitted in the original PTE. If yes, it then marks the dirty bitmap to indicate that the page has been written to during this round and extends the write permission to the shadow PTE for this round. Hence, subsequent writes to the page in this round do not cause page faults to Xen - an optimization that reduces the overhead of dirty page tracking. At the end of each round, the shadow page table infrastructure for the guest OS is dismantled and the above mentioned process continues till the stop-copy phase.

For realizing speculatively replicated VM migrations, we need to maintain a dirty bitmap per migration thread. This is because memory transfer to different destination servers may progress at different rates, leading to a distinct set of dirty pages per thread at each round. For example, a given pre-copy round page transfer may take 1 unit of time for one destination while the same round can take 2x the time for another due to a slow network path or destination. During this round the dirty pages that need to transferred in the following round is bound to vary for both the threads given the divergent round durations.

When a page fault is trapped to Xen in this mode of operation, all dirty bitmaps are updated in response. When a pre-copy round is complete for a migration thread it always dismantles the shadow paging infrastructure. This causes additional page fault traps to Xen compared to the non-replicated standard migration. For example, consider two migration threads t1 and t2, where t2 is slower than t1 in transferring pages to its destination server. When thread t1's pre-copy round is complete, it dismantles the shadow paging infrastructure. A subsequent access to page 'p' by the VM results in a trap to Xen and the PTE's write permissions from the original page table being transferred to the shadow page table. Now, if before page 'p' is written to again by the VM, if thread t2 completes its pre-copy round, it ends up dismantling the shadow paging infrastructure again. So for the same round in thread t1, a subsequent access to page 'p' will trigger another trap to Xen for the sake of dirty accounting for

thread t2. This is an inevitable consequence of replicated migration and the overhead increases with the number of simultaneously replicated migrations.

Finally, when one migration thread reaches Stage 5 of the migration transaction, all other migration replicas are issued an abort while the chosen migration commits and activates the VM. Ties are broken arbitrarily when they arise. At this point the VM state is externalized to other VMs and the cloud infrastructure. Note that callers of the speculative migration operation, can also choose to abort transactions at the end of earlier stages by using the progress rate specification as explained next.

5.3.2.3 Cost vs. Benefit of Speculative Migrations

Progress rate Based Speculation Abort

To control the above issue and the network usage overhead of replicated migrations, the speculative migration implementation offers progress rate based termination of replicas. Recall that, each migration operation is composed of a series of stages that must be executed in order, atomically, as part of a transaction abstraction. There is a natural notion of migration progress as each migration goes through the multiple stages. The user can specify when to abort all but one concurrent speculations in order to control the overhead. Early speculation aborts result in minimal overhead and correspondingly minimal fault-tolerance, whereas late speculation aborts result in higher fault-tolerance and higher overheads. The choice is left to the capacity multiplexing service which may use service specific knowledge such as employing higher fault-tolerance for more important VMs (e.g. from a class of service perspective, gold vs. bronze VMs).

Since multiple stages of the migration operation do not contribute equally to the progress of the overall migration, especially in the case of the iterative pre-copy stage which is the most intensive, we also enable the forecasting of the number of pre-copy

rounds based on observed VM dirty rates during a migration using models developed by Liu et.al. [74] and Akoush et.al. [33], for improved accuracy in calculating progress rate. Note that we do not as such require a high degree of precision in the estimates for our use case and that the dirty rate is already tracked as part of the migration process resulting in no extra overhead.

Failure Suspicion Service

In addition to the progress rate based speculative replica abort, the infrastructure management stack also supports a querying interface with which automation services like capacity multiplexing can obtain estimated failure likelihood of different servers at any point in time. This information can be used to employ speculative replication only when the estimated likelihood is higher - i.e., there is a good benefit to expending the extra cost for replicated migration.

An agent runs on each physical server and the central controller in the datacenter and computes a (0-1) likelihood of a migration failing on a particular host or along a particular network path due to bandwidth insufficiency. The network path statistics are gathered by the central controller periodically, by querying datacenter switches via ssh on a switch's management IP.

We employ an approximate continuous domain failure metric as introduced by Hayashibara et.al. [50] that captures the inherent difficulty in accurately predicting failures in a complex datacenter environment. The decreased accuracy is sufficient for the purpose of cost vs. benefit optimization where the results of mischaracterization are not catastrophic and results in some wasted resources in the case of false positives or failed migrations in the case of false negatives.

The failure suspicion module can look for a variety of indicators such as resource utilization of hosts or network links, history of operation failures and also read key

system logs like kernel and hypervisor logs to compute the failure suspicion metric. The methods used for its computation need not be expensive or sophisticated as high accuracy is not an absolute requirement in our case.

VM Migration Cost Estimator

As a final method to tailor the cost vs. benefit of speculation, the virtualized infrastructure management stack also provides the ability for capacity multiplexing services to ask for an estimate of a VM's page dirtying rate. This is implemented in the hypervisor where shadow paging is periodically turned on (say once every 5 or 10 minutes) for tracked VMs for very brief durations (tens of seconds) over the course of its execution, to estimate its page dirty rate during that brief interval. Data from multiple intervals are smoothed using a moving average filter and estimate of the VM dirtying rate is exposed to multiplexing services in order for them to roughly calculate the cost of migrating a VM either speculatively or normally or choosing a different VM altogether. This information can again used to further judiciously employ speculation.

5.3.3 Speculor

We now move on to illustrating how speculatively replicated VM migration operations can be leveraged by a capacity multiplexing solution to reduce the number of failed migrations and hence improve the performance of VMs. We do this by implementing a top-k load balancing algorithm that iteratively reduces the number of resource overloaded servers as described by Gulati et.al. [59]. The algorithm works by selecting the top 'k' most loaded hosts in a cluster and the top 'k' least loaded hosts and balancing the VM load between this set's capacity. The value 'k' is a parameter configurable by the system administrator to balance overhead and effectiveness of datacenter hotspot

reduction. Across several invocations of the algorithm, the entire cluster capacity is more appropriately allocated to match VM demands. This algorithm also has good scaling properties as it operates on a small subset of servers of a potentially large cluster. The algorithm, and our extension to it to take advantage of speculatively replicated migrations, is illustrated in Algorithm 1.

Algorithm 1: Top-k Load Balancing Algorithm

```
hosts ← SortAscending(hosts)
leastloaded ← hosts[1 to topk]
mostloaded ← hosts[n to (n - topk)]
setwideavg ← ComputeAvg(leastloaded, mostloaded)
setwidestdev ← ComputeStdev(leastloaded, mostloaded)
migrations ← [ ]
Pass 1
foreach mhost in mostloaded do
    if mhost load is within half setwidestdev of setwideavg then
        continue
    foreach vm in mhost do
        foreach lhost in leastloaded do
            if lhost does not have enough resources to run vm then
                continue
            Update lhost state with vm
            if lhost load is less or equal to half setwidestdev of setwideavg then
                foundhost = lhost break
            else
                Remove vm from lhost state
        if foundhost is not NULL then
            migration ← create_migration(vm, foundhost) Add migration to
            migrations
Pass 2
leastloaded ← SortAscending(leastloaded)
while SPEC_QUOTIENT > 0 do
    foreach migration in migrations do
        foreach lhost in leastloaded do
            if lhost already part of migration then
                continue
            if lhost does not have enough resources to run vm then
                continue
            Add lhost to migration as additional destination
    SPEC_QUOTIENT = SPEC_QUOTIENT - 1
```

Pass 1 in the listing above is the regular top-k load balancing algorithm. Here

we try to move VMs from overloaded hosts to least loaded hosts and try to bring the average resource utilization of the hosts to within half a standard deviation away from the set-wide average utilization. This results in all VMs in the set more or less equally sharing the total set resource capacity. The load metric used in the algorithm can be based on either a single resource or multiple resources like CPU and memory, computed as a weighted sum, as shown in Chapter 3.

We extend this algorithm to include an additional pass (Pass 2 in the listing) where for each selected migration, additional destination hosts, from among the least loaded hosts with spare capacity, are selected. The SPECULATIVE_QUOTIENT (SQ, for brevity) factor specified in the listing, statically governs the number of speculative replicas of migration to execute[1]. In trying to move a VM from an overloaded host to an under-loaded host, several candidates are likely more or less equally preferable. This fact is captured in the algorithm. Also, since the least loaded hosts form a random subset of the large cluster of machines, they are likely to fail independently of each other when failures are uncorrelated. So the likelihood of all speculative migrations all failing at the same time is extremely minimal.

Speculor is implemented in Python as an OpenStack service running as part of a central cloud controller. It gathers monitoring data via libvirt and triggers speculative migrations through a modified OpenStack compute API that exposes them.

5.4 Evaluation

The goal of our evaluation is to show that using speculatively replicated VM live migrations, capacity multiplexing systems can reduce the number of failed logical migrations and that the overhead of replication is acceptable compared to its benefits. We show results on a 12 server cluster and the behavior of Speculor at larger scales using simulations. We vary the fraction of failed nodes in the cluster at any point in

[1]Note that more sophisticated versions of the top-k algorithm can dynamically select the value of SQ based on information available via the management stack's querying interface.

time, and the amount of speculation, to prove our hypothesis.

5.4.0.1 Testbed

Our testbed consists of 12 4-socket, Quad-core servers (total of 16 cores) with 48GB of memory each, connected together through 1Gbps network links. The servers are all virtualized using the Xen 4.2.1 hypervisor with our replicated migration implementation. The whole cluster is managed through OpenStack (Grizzly release) with additional modifications to expose the speculatively replicated migrations via the nova API. The VMs used in our experiments are all configured with 4 virtual CPUs, 4GB of memory and 10GB of disk space. The VM disk images are stored as large files in a clustered storage environment built using GlusterFS [10] with data striped in small 128KB chunks across 1TB disks on the 12 servers. GlusterFS exposes the clustered storage as a single mount point on each server, accessible via the POSIX filesystem API. Therefore, VM live migrations do not need movement of their disk images.

5.4.0.2 Parametric Failure Simulation

In order to create a controllable failure-prone environment in which to test Speculor and its fault-scalability, we created a parametric failure simulation system that consists of a single master and multiple failure simulating agents, one per server in our cluster. The agents register with the master using a well known IP and port combination at the start of the failure simulation. The master sends periodic messages to each agent indicating if they are in a failed state or not. The agents ensure that all VM migration receive requests are rejected when they are in a failed state.

The master takes in a failure specification with the following parameters:

- Percentage of servers in failed state at any given time during the failure simulation.
- The minimum and maximum duration of each failure event on a given server.

The duration is specified in seconds. The goal here is to mimic intermittent and temporary failures as commonly seen in datacenter environments.

- Total simulation time in minutes.

The failure simulation works as follows. At the start of the simulation, the master uniformly at random picks the required percentage of failed nodes from the available servers. Each node is then assigned a failure event duration, again, chosen uniformly at random between the user specified minimum and maximum durations. It then notifies the appropriate failure simulation agents of the failure event interval. Any live migration requests to these servers will now be rejected resulting in a failure.

The master then sleeps till the first node's or set of nodes' failure events expire and picks replacement nodes to be in failed mode, using the process described above. Therefore, at any given time during the failure simulation the required percentage of failed nodes are maintained and the duration of each failure event on an individual server is variable as seen in real-life. The simulation continues till the total simulation time has expired. The algorithm to accomplish this is a straightforward greedy job scheduling algorithm where the required percentage failed nodes are the slots which need to be filled with randomly chosen nodes, with each node requiring a different amount of service as determined by the failure event duration.

5.4.0.3 Testbed Results

For this experiment we deploy 64 VMs on the 12 servers with the VMs running a scaled down version, according to the cluster size, of the Google trace workload described in Chapter 3 Section 3.4. Each set of 16 VMs now replay the resource usage pattern of each of the 4 jobs in the trace. For the sake of convenience the first two hours of the trace job utilization is replayed. At power-on time, the VMs are placed on the physical hosts evenly in terms of configured capacity as determined by OpenStack nova SimpleScheduler. Speculor's load balancing algorithm is run once

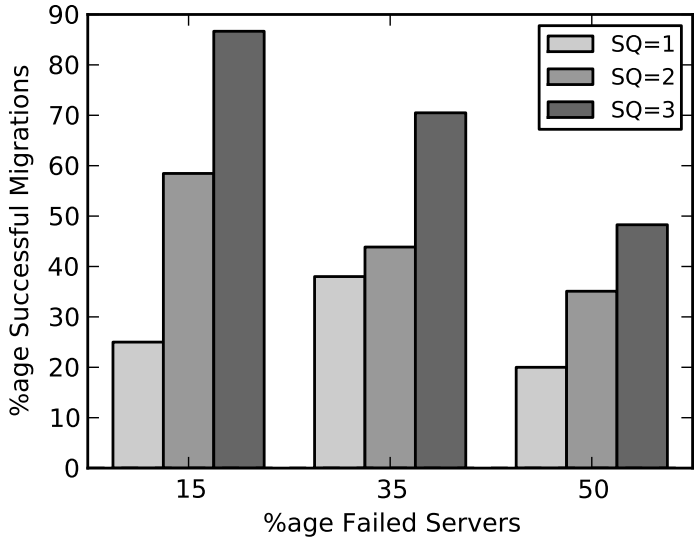


Figure 30: Fraction of successful migrations under different failure and speculative replication intensities.

every 5 minutes in this scenario over a total of 2 hours. We specify the 'k' value of the algorithm, in percentage, as 35%. In other words, the top 4 most loaded and least hosts are considered by the algorithm during each invocation. The failure event durations for this experiment vary between 30 and 120 seconds.

Figure 30 shows the fraction of successful migrations for increasing percentage of failed nodes under different levels of speculative replication. The amount of speculative replication is indicated by the SQ variable in the graphs. SQ equal to 1 refers to non-speculative standard migration. It can be seen that increasing the amount of speculative replication results in higher migration success rates for each of the given percentage of failed nodes; over 200% in the case of 15% failed hosts. Additionally, it can also be seen that as the percentage of failed nodes increase, i.e., from 15% to 50%, the gains from speculatively replicating migration operations decrease accordingly to almost half the effectiveness under low failure rates. In a small cluster as the failure rate increases, there are not enough hosts to hedge the logical migration against.

In terms of the overheads of speculative migrations, the predominant increase

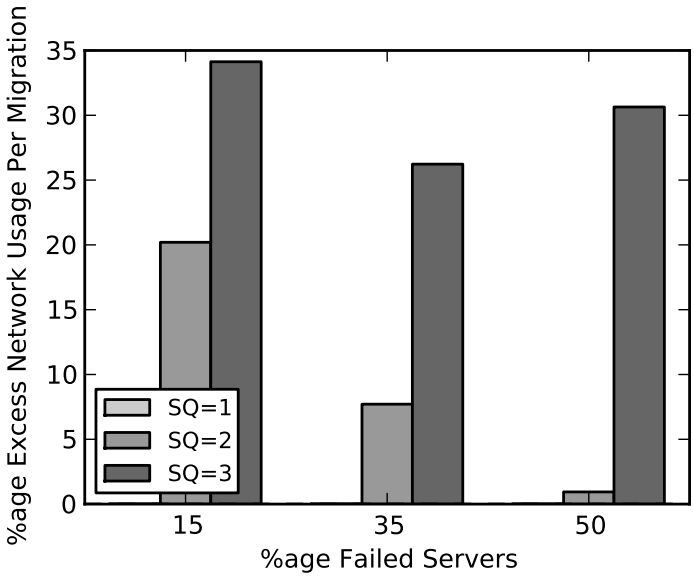


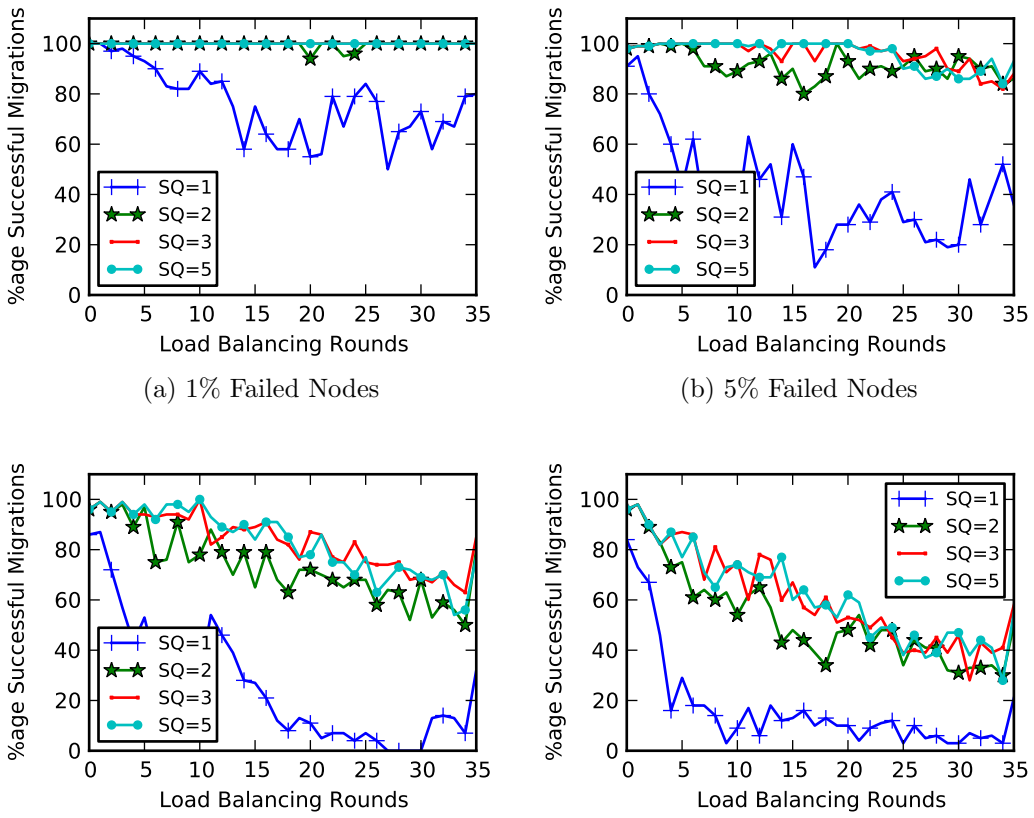
Figure 31: Normalized excess network usage per successful VM migration (replicated or otherwise).

is in the consumption of cluster network bandwidth for the composite migration operation, in comparison with non-replicated migration. Figure 31 shows the excess bandwidth used for different amounts of speculation normalized to the non-speculative VM migration (i.e., SQ=1, which is reported as zero in the graph). Because of the way the failure simulation is setup, i.e., to reject all migration receive calls for a failed hosts, the failed speculative migrations themselves do not consume any resources. However, if several replicated migrations do not fail and all reach near completion, then the network usage overhead can be expected to be magnified as: a multiple of the overhead of a single migration times the number of migration replicas. This situation arises when excessive speculation is used while cluster failure rates are low (e.g. SQ=3 and failure rate=15%).

In addition, we used a simple progress rate based speculation termination where progression through each of the migration stages contributes equally to the overall progress rate of the operation. We set this value to 30% or abort all speculation after Stage 1, base image transfer is complete, of the winning replica. This further

reduces network overhead due to subsequent rounds. Therefore, the network overhead of speculation is minimal compared to the gains achieved in number of successful migrations.

5.4.0.4 Simulation Results



(a) 1% Failed Nodes

(b) 5% Failed Nodes

(c) 10% Failed Nodes

(d) 15% Failed Nodes

Figure 32: Fraction of successful migrations at each round under different failure and speculative replication intensities.

To better study the behavior of Speculor's algorithm at large scale, we build a datacenter discrete event simulator that can create racks, servers and different network topologies. Different server hardware configurations, in terms of CPU and memory capacity, can also be specified. The simulator can power on VMs of different flavors (configurations obtained from Amazon EC2 instance types [1]) on to the physical servers in random. For the simulation results presented here, we use an equal number

of VMs of all flavors. Each of the powered-on VMs have ongoing CPU and memory demand, specified in percentage, drawn from a normal distribution. The simulator also includes the parametric failure simulation logic explained in the previous subsection. The minimum and maximum failure event duration values we use here are 30 and 180 seconds, respectively.

We also implement replicated and standard VM live migration operations in the simulator, and, the Speculor top-k load balancing algorithm. Note that we only simulate events in our datacenter i.e., VM power-ons, load balancing algorithm invocations and live migrations.

We specify a total of 1024 servers of 3 different configurations reflecting small, medium and large hosts in terms of resource capacity. We initially power on 2000 VMs on the entire cluster and then run the Speculor load balancing algorithm on this environment to dynamically place VMs on different physical servers (through live migrations) as their resource demands change over time. We use a 'k' value in the algorithm, specified in percentage as before, as 10%. In other words the top 102 most loaded and least loaded servers are considered by the algorithm during each invocation. For all of the configurations reported next, we run the algorithm for the same number of rounds.

Figure 32 shows the percentage of successful VM migrations for each round of load balancing, for different failed node fractions and amounts of speculative replication. An important trend that can be seen across the graphs is that a limited amount of speculative replication, i.e., SQ=2, performs almost as good as higher levels of speculative replication in large scale environments. Intuitively, for the same failure fraction, this is because when an additional host is chosen for replicating a migration to, from a larger pool of hosts, the likelihood of both the hosts involved in the replicated migration being in a failed state is much smaller compared a smaller scale

cluster. This is true when failures are uncorrelated as is the common case in datacenters. On average, replicating the live migration operation on one additional host, yields between 31% to over 200% improvement in number of successful migrations with the benefit increasing with higher failure rates.

Therefore, one can draw the conclusion that even a limited amount of speculative replication of migrations is enough to achieve substantial gains in large scale computing environments. With additional overhead reducing techniques such as progress rate based termination and providing consumers of the speculative API with information on the cost (i.e., VM memory dirty rates) and importance (i.e., failure suspicion) of applying speculation, we believe that speculative replication can be used on an ongoing basis in even production datacenters to better achieve capacity multiplexing objectives.

5.5 Related Work

Several studies exist that present data on service failures in datacenter environments and on application design for this environment. Oppenheimer et.al., analyzed Internet service failure rates from three hosting centers and conclude that configuration errors by administrators is the largest cause for service failure and associated downtime [84]. Similar conclusions are also corroborated by Hamilton et.al. [62]. They also advocate using application level techniques to tolerating failures and service liveness checking and exclusion. While application level techniques for dealing with infrastructure related failures and poor performance are useful, they typically add to the complexity of development. The employed methods need to be always "on", given the absence of infrastructure level information, and a certain performance penalty is paid throughout the application's lifetime.

The Recovery-Oriented Computing project [87] takes the view that failures in

hardware and software due to a variety of reasons are inevitable in large scale computing environments. It is a view we share as well. However, instead of advocating that applications be re-designed for this environment, we believe that some level of fault-tolerance and fault-scalability needs to be built into the infrastructure itself to avoid the above mentioned problems.

In addition to fault-tolerant application designs, single and distributed system root-cause diagnosis systems have also been employed to deal with failures in large scale environments, Aguilera et.al. [32] log messages between components of distributed applications and find causal relationships between them in order to identify problematic nodes and components of the distributed application. The X-trace [53] system has similar goals for distributed applications where each system request is tagged and its path through the software and protocol stack is traced to present a detailed view of performance problems. Their system requires modifications all the way along the software stack, making it cumbersome, and in scenarios is incapable of finding out the appropriate root causes. The SNAP [98] system tackles the problem of network performance diagnosis by collecting application level socket API calls and TCP statistics. It then correlates them across datacenter servers and switches to identify performance problems. The X-ray [35] system finds problems in single application binaries by dynamically instrumenting them and attributing performance problems to either configuration or input values

Overall, given the varied failure modes we are also of the opinion that pinpointing the root causes underlying them in a large scale distributed system, is a time consuming and inaccurate process to be used in an ongoing basis. It also requires manual intervention to develop and deploy patches to fix identified problems. We envision a speculative replication system such as ours as complimentary to root-cause diagnosis systems and more usable on a regular basis in an imperfect operating environment.

5.6 Chapter Summary

In this chapter we presented the design and implementation of speculatively replicated VM live migration operation and how it can be leveraged by capacity multiplexing systems to minimize a wide variety of failures with minimal overhead. The design features a clean separation of concerns in the implementation and semantics of speculative replication, in the virtualized infrastructure management stack, and, its use (policy) by higher layer capacity multiplexing systems. The results show that in large scale environments even a small amount of speculative replication can result in large gains in fault-tolerance and performance. Our design is general enough to be realized for other operations on virtualized entities like VMs and virtual networks which may be beneficial for common datacenter automation systems such as auto-scaling, application orchestration and deployment, server software maintenance via rolling upgrades etc.

CHAPTER VI

CONCLUSIONS AND FUTURE WORK

In this thesis we have presented virtualized datacenter compute capacity multiplexing systems for large scale environments that focus on achieving scalability through simple, easy to reason about methods, and robustness in a failure-agnostic way. We propounded the idea that in designing allocation systems for the reality of variable actuation costs, infrastructure resource limitations and prevailing failure behavior, requires non-intuitive tradeoffs in accuracy and complexity.

We presented CCM, a hierarchically organized, on-demand compute capacity multiplexing system that achieves scalability through the use of various low overhead techniques developed by analyzing publicly available trace data from production datacenters and via an iterative refinement process that involved running the system on a 700 server datacenter. We developed the Xerxes distributed load generation framework to exercise the system in a number of real-life and anticipated scenarios. Representative cloud applications representing popular class of cloud codes were also used to quantify CCM's benefits. CCM enables improved capacity multiplexing performance, in terms of resource availability for VMs, with reduced overhead proportional to the amount of oversubscription and variability inherent in the workload.

The data and insights on VM live migration failures obtained from experiments with CCM, guided the design of Speculor, a statistically scalable capacity multiplexing algorithm that employs speculative replication of VM live migration operations to achieve fault-scalable performance in the presence of a wide variety of failures. The results indicate that even a limited amount of speculative replication can produce significant gains in capacity multiplexing performance with minimal overhead.

In the future, we will explore the complete design of a virtualized management stack that supports speculatively replicated operations on all virtualized entities like VMs, virtual networks and virtual storage, and the re-design of most datacenter automation services like auto-scaling, application deployment and orchestration, rolling upgrades etc. We will also explore tolerating correlated failure scenarios in datacenters caused by operator mistakes, for example, through the use of "correlated failure domains" identified automatically or through administrator input.

REFERENCES

[1] "Amazon EC2 instances." http://aws.amazon.com/ec2/instance-types/.

[2] "Amazon elastic compute cloud (EC2)." http://aws.amazon.com/ec2/.

[3] "Apache Nutch." http://nutch.apache.org/.

[4] "Cloud Dashboard — Ravello Systems." http://www.ravellosystems.com/cloud-dashboard.

[5] "Cloud Dashboard Part 1 - VM Provisioning." http://www.ravellosystems.com/blog/cloud-dashboard-part-1-vm-provisioning/.

[6] "Configuration Maximums - VMware vSphere 5.1." http://pubs.vmware.com/vsphere-51/index.jsp.

[7] "DRS Performance and Best Practices." http://www.vmware.com/resources/techresources/1062.

[8] "Error Retries and Exponential Backoff in AWS." http://docs.aws.amazon.com/general/latest/gr/api-retries.html.

[9] "Facebook: Ship early and ship twice as often." https://www.facebook.com/notes/facebook-engineering/ship-early-and-ship-twice-as-often/10150985860363920.

[10] "GlusterFS." http://www.gluster.org.

[11] "googleclusterdata - Traces of Google tasks running in a production cluster." http://code.google.com/p/googleclusterdata/.

[12] "HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers." http://www.netlib.org/benchmark/hpl/.

[13] "httperf." http://www.hpl.hp.com/research/linux/httperf/.

[14] "Hyper-V: Using Hyper-V and Failover Clustering." http://technet.microsoft.com/en-us/library/cc732181(v=ws.10).aspx.

[15] "ISTC-CC Research: Benchmarks." http://www.istc-cc.cmu.edu/research/benchmarks/index.shtml.

[16] "LAPACK - Linear Algebra Package." http://www.netlib.org/lapack/.

[17] "OpenStack - Open Source Cloud Computing Software." https://www.openstack.org/.

[18] "Project Voldemort." http://project-voldemort.com/.

[19] "Public cloud hosting, computing, storage and networking by Rackspace." http://www.rackspace.com/cloud/.

[20] "Rackspace Hosting - Cloud Terms of Service." http://www.rackspace.com/information/legal/cloud/tos.

[21] "Release Cycle - OpenStack." https://wiki.openstack.org/wiki/Release_Cycle.

[22] "Unified Computing System." http://www.cisco.com/en/US/netsol/ns944/index.html.

[23] "vCloud Suite - Infrastructure-as-a-Service and Cloud Computing." http://www.vmware.com/products/vcloud-suite.

[24] "VMware Distributed Power Management Concepts and Use." http://www.vmware.com/files/pdf/DPM.pdf.

[25] "VMware DRS." http://www.vmware.com/products/DRS.

[26] "VMware VI (vSphere) Java API." http://vijava.sourceforge.net/.

[27] "VMware vSphere." http://www.vmware.com/products/vsphere/.

[28] "vSphere High Availability (HA) - Technical Deepdive." http://www.yellow-bricks.com/vmware-high-availability-deepdiv.

[29] "Xen Cloud Platform Administrator's Guide - Release 0.1," 2009.

[30] "Citrix Workload Balancing 2.1 Administrator's Guide," 2011.

[31] "Cloud Infrastructure Architecture Case Study," 2012. http://www.vmware.com/resources/techresources/10255.

[32] Aguilera, M. K., Mogul, J. C., Wiener, J. L., Reynolds, P., and Muthitacharoen, A., "Performance Debugging for Distributed Systems of Black Boxes," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, (New York, NY, USA), pp. 74–89, ACM, 2003.

[33] Akoush, S., Sohan, R., Rice, A., Moore, A. W., and Hopper, A., "Predicting the Performance of Virtual Machine Migration," in *Proceedings of the 2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, MASCOTS '10, (Washington, DC, USA), pp. 37–46, IEEE Computer Society, 2010.

[34] Al-Fares, M., Loukissas, A., and Vahdat, A., "A Scalable, Commodity Data Center Network Architecture," in *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, SIGCOMM '08, (New York, NY, USA), pp. 63–74, ACM, 2008.

[35] Attariyan, M., Chow, M., and Flinn, J., "X-ray: Automating root-cause diagnosis of performance anomalies in production software," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, (Berkeley, CA, USA), pp. 307–320, USENIX Association, 2012.

[36] Babu, S., "Towards Automatic Optimization of MapReduce Programs," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, (New York, NY, USA), pp. 137–142, ACM, 2010.

[37] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A., "Xen and the Art of Virtualization," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, (New York, NY, USA), pp. 164–177, ACM, 2003.

[38] Barroso, L. A. and Hölzle, U., *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Synthesis Lectures on Computer Architecture, Morgan & Claypool Publishers, 2009.

[39] Bodik, P., Fox, A., Franklin, M. J., Jordan, M. I., and Patterson, D. A., "Characterizing, modeling, and generating workload spikes for stateful services," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, (New York, NY, USA), ACM, 2010.

[40] Brian F. Cooper, Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R., "Benchmarking cloud serving systems with YCSB," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, (New York, NY, USA), ACM, 2010.

[41] Cattell, R., "Scalable SQL and NoSQL data stores," *SIGMOD Rec.*, vol. 39, pp. 12–27, May 2011.

[42] Chen, M. Y., Kiciman, E., Fratkin, E., Fox, A., and Brewer, E., "Pinpoint: Problem Determination in Large, Dynamic Internet Services," in *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, DSN '02, (Washington, DC, USA), pp. 595–604, IEEE Computer Society, 2002.

[43] Chen, M., Zhang, H., Su, Y.-Y., Wang, X., Jiang, G., and Yoshihira, K., "Effective VM sizing in virtualized data centers," in *Integrated Network Management (IM), 2011 IFIP/IEEE International Symposium on*, pp. 594–601, May 2011.

[44] Chen, Y., Ganapathi, A. S., Griffith, R., and Katz, R. H., "Analysis and Lessons from a Publicly Available Google Cluster Trace," Tech. Rep. UCB/EECS-2010-95, EECS Department, University of California, Berkeley, Jun 2010.

[45] Clark, C., Fraser, K., Hand, S., Hansen, J. G., Jul, E., Limpach, C., Pratt, I., and Warfield, A., "Live migration of virtual machines," in *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, (Berkeley, CA, USA), pp. 273–286, USENIX Association, 2005.

[46] Dean, J. and Barroso, L. A., "The Tail at Scale," *Commun. ACM*, vol. 56, pp. 74–80, Feb. 2013.

[47] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W., "Dynamo: Amazon's Highly Available Key-value Store," in *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, (New York, NY, USA), pp. 205–220, ACM, 2007.

[48] Delimitrou, C., Sankar, S., Vaid, K., and Kozyrakis, C., "Decoupling Datacenter Studies from Access to Large-scale Applications: A Modeling Approach for Storage Workloads," in *Proceedings of the 2011 IEEE International Symposium on Workload Characterization*, IISWC '11, (Washington, DC, USA), pp. 51–60, IEEE Computer Society, 2011.

[49] Do, T., Hao, M., Leesatapornwongsa, T., Patana-anake, T., and Gunawi, H. S., "Limplock: Understanding the Impact of Limpware on Scale-out Cloud Systems," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, (New York, NY, USA), pp. 14:1–14:14, ACM, 2013.

[50] Dfago, X., Urbn, P., Hayashibara, N., and Katayama, T., "The ϕ accrual failure detector," in *RR IS-RR-2004-010, Japan Advanced Institute of Science and Technology*, pp. 66–78, 2004.

[51] Fagg, G. E. and Dongarra, J., "FT-MPI: Fault Tolerant MPI, Supporting Dynamic Applications in a Dynamic World," in *Proceedings of the 7th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, (London, UK, UK), pp. 346–353, Springer-Verlag, 2000.

[52] Ferdman, M., Adileh, A., Kocberber, O., Volos, S., Alisafaee, M., Jevdjic, D., Kaynak, C., Popescu, A. D., Ailamaki, A., and Falsafi, B., "Clearing the clouds: a study of emerging scale-out workloads on modern hardware," in *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '12, (New York, NY, USA), pp. 37–48, ACM, 2012.

[53] Fonseca, R., Porter, G., Katz, R. H., Shenker, S., and Stoica, I., "X-trace: A Pervasive Network Tracing Framework," in *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation*, NSDI'07, (Berkeley, CA, USA), pp. 20–20, USENIX Association, 2007.

[54] Ford, D., Labelle, F., Popovici, F. I., Stokely, M., Truong, V.-A., Barroso, L., Grimes, C., and Quinlan, S., "Availability in Globally Distributed Storage Systems," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, (Berkeley, CA, USA), pp. 1–7, USENIX Association, 2010.

[55] Ghemawat, S., Gobioff, H., and Leung, S.-T., "The Google File System," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, (New York, NY, USA), pp. 29–43, ACM, 2003.

[56] Gmach, D., Rolia, J., Cherkasova, L., and Kemper, A., "Workload Analysis and Demand Prediction of Enterprise Data Center Applications," in *Proceedings of the 2007 IEEE 10th International Symposium on Workload Characterization*, IISWC '07, (Washington, DC, USA), pp. 171–180, IEEE Computer Society, 2007.

[57] Greenberg, A. and others, "VL2: a scalable and flexible data center network," in *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, SIGCOMM '09, (New York, NY, USA), pp. 51–62, ACM, 2009.

[58] Greenberg, A., Hamilton, J., Maltz, D. A., and Patel, P., "The cost of a cloud: Research problems in data center networks," *SIGCOMM Comput. Commun. Rev.*, vol. 39, pp. 68–73, Dec. 2008.

[59] Gulati, A. and others, "Cloud-scale resource management: challenges and techniques," in *Proceedings of the 3rd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'11, (Berkeley, CA, USA), pp. 3–3, USENIX Association, 2011.

[60] Gulati, A. and others, "Decentralized management of virtualized hosts," 12 2012. Patent No. US20120324441 A1.

[61] Gulati, A. and others, "VMware Distributed Resource Management: Design, implementation and lessons learned," *VMware Technical Journal*, vol. 1, pp. 45–64, Apr 2012.

[62] Hamilton, J., "On designing and deploying internet-scale services," in *Proceedings of the 21st Conference on Large Installation System Administration Conference*, LISA'07, (Berkeley, CA, USA), pp. 18:1–18:12, USENIX Association, 2007.

[63] Hoffmann, H., Sidiroglou, S., Carbin, M., Misailovic, S., Agarwal, A., and Rinard, M., "Dynamic knobs for responsive power-aware computing,"

in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, (New York, NY, USA), pp. 199–212, ACM, 2011.

[64] Huang, S., Huang, J., Dai, J., Xie, T., and Huang, B., "The HiBench benchmark suite: Characterization of the MapReduce-based data analysis," *2013 IEEE 29th International Conference on Data Engineering Workshops (ICDEW)*, vol. 0, pp. 41–51, 2010.

[65] Isard, M., "Autopilot: automatic data center management," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 2, pp. 60–67, 2007.

[66] Jiang, W., Hu, C., Zhou, Y., and Kanevsky, A., "Are Disks the Dominant Contributor for Storage Failures?: A Comprehensive Study of Storage Subsystem Failure Characteristics," *Trans. Storage*, vol. 4, pp. 7:1–7:25, Nov. 2008.

[67] Jin, G., Song, L., Shi, X., Scherpelz, J., and Lu, S., "Understanding and Detecting Real-world Performance Bugs," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, (New York, NY, USA), pp. 77–88, ACM, 2012.

[68] Jung, G., Joshi, K. R., Hiltunen, M. A., Schlichting, R. D., and Pu, C., "A cost-sensitive adaptation engine for server consolidation of multitier applications," in *Middleware '09*.

[69] Krieger, O., McGachey, P., and Kanevsky, A., "Enabling a marketplace of clouds: VMware's vCloud director," *SIGOPS Oper. Syst. Rev.*, vol. 44, pp. 103–114, December 2010.

[70] Kumar, S., Talwar, V., Kumar, V., Ranganathan, P., and Schwan, K., "vmanage: loosely coupled platform and virtualization management in data centers," in *Proceedings of the 6th International Conference on Autonomic Computing*, ICAC '09, (New York, NY, USA), pp. 127–136, ACM, 2009.

[71] Kumar, V., Cooper, B. F., Eisenhauer, G., and Schwan, K., "iManage: policy-driven self-management for enterprise-scale systems," in *Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware*, Middleware '07, (New York, NY, USA), pp. 287–307, Springer-Verlag New York, Inc., 2007.

[72] Li, A., Yang, X., Kandula, S., and Zhang, M., "CloudCmp: Comparing Public Cloud Providers," in *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, IMC '10, (New York, NY, USA), pp. 1–14, ACM, 2010.

[73] Lim, H. C., Babu, S., Chase, J. S., and Parekh, S. S., "Automated control in cloud computing: challenges and opportunities," in *ACDC '09*.

[74] Liu, H., Xu, C.-Z., Jin, H., Gong, J., and Liao, X., "Performance and Energy Modeling for Live Migration of Virtual Machines," in *Proceedings of the 20th International Symposium on High Performance Distributed Computing*, HPDC '11, (New York, NY, USA), pp. 171–182, ACM, 2011.

[75] Malkowski, S., Hedwig, M., and Pu, C., "Experimental evaluation of N-tier systems: Observation and analysis of multi-bottlenecks," *IEEE Workload Characterization Symposium*, vol. 0, pp. 118–127, 2009.

[76] Mao, M. and Humphrey, M., "A Performance Study on the VM Startup Time in the Cloud," in *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pp. 423–430, June 2012.

[77] Massie, M. L., Chun, B. N., and Culler, D. E., "The ganglia distributed monitoring system: design, implementation, and experience.," *Parallel Computing*, vol. 30, 2004.

[78] Melo, M., Maciel, P., Araujo, J., Matos, R., and Araujo, C., "Availability Study on Cloud Computing Environments: Live Migration As a Rejuvenation Mechanism," in *Proceedings of the 2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, DSN '13, (Washington, DC, USA), pp. 1–6, IEEE Computer Society, 2013.

[79] Meng, X., Isci, C., Kephart, J., Zhang, L., Bouillet, E., and Pendarakis, D., "Efficient resource provisioning in compute clouds via vm multiplexing," in *Proceedings of the 7th International Conference on Autonomic Computing*, ICAC '10, (New York, NY, USA), pp. 11–20, ACM, 2010.

[80] Moore, J. and Chase, J., "Data center workload monitoring, analysis, and emulation," in *in Eighth Workshop on Computer Architecture Evaluation using Commercial Workloads*, 2005.

[81] Moreno-Vozmediano, R., Montero, R. S., and Llorente, I. M., "Elastic management of cluster-based services in the cloud," in *Proceedings of the 1st Workshop on Automated Control for Datacenters and Clouds*, ACDC '09, (New York, NY, USA), pp. 13–18, ACM, 2009.

[82] Nathuji, R., Kansal, A., and Ghaffarkhah, A., "Q-clouds: managing performance interference effects for qos-aware clouds," in *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, (New York, NY, USA), pp. 237–250, ACM, 2010.

[83] Niranjan Mysore, R. and others, "Portland: a scalable fault-tolerant layer 2 data center network fabric," in *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, SIGCOMM '09, (New York, NY, USA), pp. 39–50, ACM, 2009.

[84] Oppenheimer, D., Ganapathi, A., and Patterson, D. A., "Why Do Internet Services Fail, and What Can Be Done About It?," in *Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems - Volume 4*, USITS'03, (Berkeley, CA, USA), pp. 1–1, USENIX Association, 2003.

[85] Padala, P., Shin, K. G., Zhu, X., Uysal, M., Wang, Z., Singhal, S., Merchant, A., and Salem, K., "Adaptive control of virtualized resources in utility computing environments," in *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, (New York, NY, USA), pp. 289–302, ACM, 2007.

[86] Patil, S., Polte, M., Ren, K., Tantisiriroj, W., Xiao, L., López, J., Gibson, G., Fuchs, A., and Rinaldi, B., "YCSB++: Benchmarking and Performance Debugging Advanced Features in Scalable Table Stores," in *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, SOCC '11, (New York, NY, USA), pp. 9:1–9:14, ACM, 2011.

[87] Patterson, D., Brown, A., Broadwell, P., Candea, G., C, G., Chen, M., Cutler, J., Armando, Enriquez, P., Fox, O., Oppenheimer, D., Kcman, E., Merzbacher, M., Sastry, N., Tetzlaff, W., Traupman, J., and Treuhaft, N., "Recovery Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies," tech. rep., Berkeley Computer Science, 2002.

[88] Ristenpart, T., Tromer, E., Shacham, H., and Savage, S., "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, (New York, NY, USA), pp. 199–212, ACM, 2009.

[89] Sharma, S., Staessens, D., Colle, D., Pickavet, M., and Demeester, P., "Enabling fast failure recovery in OpenFlow networks," in *Design of Reliable Communication Networks (DRCN), 2011 8th International Workshop on the*, pp. 164–171, Oct 2011.

[90] Shen, Z., Subbiah, S., Gu, X., and Wilkes, J., "CloudScale: Elastic Resource Scaling for Multi-Tenant Cloud Systems," in *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SoCC '11, (New York, NY, USA), ACM, 2011.

[91] Singh, R., Sharma, U., Cecchet, E., and Shenoy, P., "Autonomic mix-aware provisioning for non-stationary data center workloads," in *Proceedings of the 7th International Conference on Autonomic Computing*, ICAC '10, (New York, NY, USA), pp. 21–30, ACM, 2010.

[92] Sobel, W., Subramanyam, S., Sucharitakul, A., Nguyen, J., Wong, H., Klepchukov, A., Patil, S., Fox, O., and Patterson, D., "Cloudstone: Multi-platform, multi-language benchmark and measurement tools for Web 2.0," in *Proceedings of Cloud Computing and Its Applications*, vol. 8, 2008.

[93] Soundararajan, V. and Anderson, J. M., "The impact of management operations on the virtualized datacenter," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, (New York, NY, USA), pp. 326–337, ACM, 2010.

[94] Verma, A., Kumar, G., and Koller, R., "The cost of reconfiguration in a cloud," in *Proceedings of the 11th International Middleware Conference Industrial Track*, Middleware Industrial Track '10, (New York, NY, USA), pp. 11–16, ACM, 2010.

[95] Waldspurger, C. A., "Memory resource management in VMware ESX server," in *Proceedings of the 5th Symposium on Operating Systems Design and implementation*, OSDI '02, (New York, NY, USA), pp. 181–194, ACM, 2002.

[96] Wood, T., Shenoy, P., Venkataramani, A., and Yousif, M., "Black-box and gray-box strategies for virtual machine migration," in *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation*, NSDI'07, (Berkeley, CA, USA), pp. 17–17, USENIX Association, 2007.

[97] Wundsam, A., Levin, D., Seetharaman, S., and Feldmann, A., "OFRewind: Enabling Record and Replay Troubleshooting for Networks," in *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'11, (Berkeley, CA, USA), pp. 29–29, USENIX Association, 2011.

[98] Yu, M., Greenberg, A., Maltz, D., Rexford, J., Yuan, L., Kandula, S., and Kim, C., "Profiling Network Performance for Multi-tier Data Center Applications," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, (Berkeley, CA, USA), pp. 5–5, USENIX Association, 2011.

[99] Zhu, Q. and Agrawal, G., "Resource Provisioning with Budget Constraints for Adaptive Applications in Cloud Environments," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, (New York, NY, USA), pp. 304–307, ACM, 2010.

[100] Zhu, X., Young, D., Watson, B. J., Wang, Z., Rolia, J., Singhal, S., Mckee, B., Hyser, C., Gmach, D., Gardner, R., Christian, T., and Cherkasova, L., "1000 islands: Integrated capacity and workload management for the next generation data center," *Cluster Computing*, vol. 12, pp. 45–57, Mar. 2009.

VITA

Mukil Kesavan hails from the port city of Chennai, India. He graduated with a Bachelor of Engineering in Computer Science and Engineering from the College of Engineering - Guindy, Anna University in Chennai, India. He received the M.S. in Computer Science degree at Georgia Institute of Technology in 2008. He completed the PhD degree in Computer Science at Georgia Institute of Technology in May 2014 advised by Dr. Karsten Schwan and Dr. Ada Gavrilovska. His research at Georgia Tech focused on scalable, fault-tolerant datacenter capacity allocation methods and on I/O performance isolation solutions for virtualized servers.