Cluster Scheduling and Management for Large-Scale Compute Clouds

Mina Sedaghat

PhD Thesis, 2015
Department of Computing Science
Umeå University
Sweden

Department of Computing Science
Umeå University
SE-901 87 Umeå, Sweden

mina@cs.umu.se

ISBN 978-91-7601-389-2
ISSN 0348-0542
UMINF 15.19

Printed by Print & Media, Umeå University, 2015

Abstract

Cloud computing has become a powerful enabler for many IT services and new technologies. It provides access to an unprecedented amount of resources in a fine-grained and on-demand manner. To deliver such a service, cloud providers should be able to efficiently and reliably manage their available resources. This becomes a challenge for the management system as it should handle a large number of heterogeneous resources under diverse workloads with fluctuations. In addition, it should also satisfy multiple operational requirements and management objectives in large scale data centers.

Autonomic computing techniques can be used to tackle cloud resource management problems. An autonomic system comprises of a number of *autonomic elements*, which are capable of automatically organizing and managing themselves rather than being managed by external controllers. Therefore, they are well suited for decentralized control, as they do not rely on a centrally managed state. A decentralized autonomic system benefits from parallelization of control, faster decisions and better scalability. They are also more reliable as a failure of one will not affect the operation of the others, while there is also a lower risk of having faulty behaviors on all the elements, all at once. All these features are essential requirements of an effective cloud resource management.

This thesis investigates algorithms, models, and techniques to autonomously manage jobs, services, and virtual resources in a cloud data center. We introduce a decentralized resource management framework, that automates resource allocation optimization and service consolidation, reliably schedules jobs considering probabilistic failures, and dynamicly scales and repacks services to achieve cost efficiency.

As part of the framework, we introduce a decentralized scheduler that provides and maintains durable allocations with low maintenance costs for data centers with dynamic workloads. The scheduler assigns resources in response to virtual machine requests and maintains the packing efficiency while taking into account migration costs, topological constraints, and the risk of resource contention, as well as fluctuations of the background load.

We also introduce a scheduling algorithm that considers probabilistic failures as part of the planning for scheduling. The aim of the algorithm is to achieve an overall job reliability, in presence of correlated failures in a data center. To do so, we study the impacts of stochastic and correlated failures on job reliability in a virtual data center. We specifically focus on correlated failures caused by power outages or failure of network components on jobs running large number of replicas of identical tasks.

Additionally, we investigate the trade-offs between vertical and horizontal scaling. The result of the investigations is used to introduce a repacking technique to automatically manage the capacity required by an elastic service. The repacking technique combines the benefits of both scaling strategies to improve its cost-efficiency.

Sammanfattning på svenska

Datormoln har kommit att bli kraftfulla möjliggörare för många nya IT-tjänster. De ger tillgång till mycket storskaliga datorresurser på ett finkornigt och omedelbart sätt. För att tillhandahålla sådana resurser krävs att de underliggande datorcentren kan hantera sina resurser på ett tillförlitligt och effektivt sätt. Frågan hur man ska designa deras resurshanteringssystem är en stor utmaning då de ska kunna hantera mycket stora mängder heterogena resurser som i sin tur ska klara av vitt skilda typer av belastning, ofta med väldigt stora variationer över tid. Därtill ska de typiskt kunna möta en mängd olika krav och målsättningar för hur resurserna ska nyttjas.

Autonomiska system kan med fördel användas för att realisera sådana system. Ett autonomt system innehåller ett antal autonoma element som automatiskt kan organisera och hantera sig själva utan stöd av externa regulatorer. Förmågan att hantera sig själva gör dem mycket lämpliga som komponenter i distribuerade system, vilka i sin tur kan bidra till snabbare beslutsprocesser, bättre skalbarhet och högre feltolerans.

Denna avhandling fokuserar på algoritmer, modeller och tekniker för autonom hantering av jobb och virtuella resurser i datacenter. Vi introducerar ett decentraliserat resurshanteringssystem som automatiserar resursallokering och konsolidering, schedulerar jobb tillförlitligt med hänsyn till korrelerade fel, samt skalar resurser dynamiskt för att uppnå kostnadseffektivitet.

Som en del av detta ramverk introducerar vi en decentraliserad schedulerare som allokerar resurser med hänsyn till att tagna beslut ska vara bra för lång tid och ge låga resurshanteringskostnader för datacenter med dynamisk belastning. Scheduleraren allokerar virtuella maskiner utifrån aktuell belastning och upprätthåller ett effektivt nyttjande av underliggande servrar genom att ta hänsyn till migrationskostnader, topologiska bivillkor och risk för överutnyttjande.

Vi introducerar också en resursallokeringsalgoritm som tar hänsyn till korrelerade fel som ett led i planeringen. Avsikten är att kunna uppnå specificerade tillgänglighetskrav för enskilda tjänster trots uppkomst av korrelerade fel. Vi fokuserar främst på korrelerade fel som härrör från problem med elförsörjning och från felande nätverkskomponenter samt deras påverkan på jobb bestående av många identiska del-jobb.

Slutligen studerar vi även hur man bäst ska kombinera horisontell och vertikal skalning av resurser. Resultatet är en process som ökar kostnadseffektivitet genom att kombinera de två metoderna och därtill emellanåt förändra fördelning av storlekar på virtuella maskiner.

Preface

This thesis contains an introductory chapter, a brief discussion on resource management challenges in cloud data centers, and the following papers:

Paper I	Mina Sedaghat, Francisco Hernández, and Erik Elmroth. Unifying Cloud Management: Towards Overall Governance of Business Level Objectives. *In Proceedings of the 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2011)*, pp. 591-597, 2011.
Paper II	Mina Sedaghat, Francisco Hernández, and Erik Elmroth. Autonomic Resource Allocation for Cloud Data Centers: A Peer to Peer Approach. *The ACM Cloud and Autonomic Computing Conference (ICCAC'14), pp. 131-140, 2014.*
Paper III	Mina Sedaghat, Francisco Hernández, Erik Elmroth, and Sarunas Girdzijauskas. Divide the Task, Multiply the Outcome: Cooperative VM Consolidation. *Extended version of a paper published in Proceedings of the 6th IEEE International Conference on Cloud Computing Technology and Science (CloudCom 2014), pp. 300-305, 2014.*
Paper IV	Mina Sedaghat, Francisco Hernández, and Erik Elmroth. Decentralized Cloud Datacenter Reconsolidation through Emergent and Topology-aware Behaviour. *Future Generation Computer Systems, Elsevier, Vol. 56, pp. 51-63, 2016.*
Paper V	Mina Sedaghat, Francisco Hernández, and Erik Elmroth. A Virtual Machine Re-packing Approach to the Horizontal vs. Vertical Elasticity Trade-off for Cloud Autoscaling. *In Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference (CAC'13)*, Article no. 6, 2013.
Paper VI	Mina Sedaghat, Eddie Wadbro, John Wilkes, Sara De Luna, Oleg Seleznjev, and Erik Elmroth. Die-Hard: Reliable Scheduling to Survive Correlated Failures in a Cloud Datacenter. *Submitted for publication, 2015.*

Financial support for this work is provided in part by the European Commission's Seventh Framework Programme [FP7/2001-2013] under grant agreement number 257115, (OPTIMIS), the European Union's Seventh Framework Programme under grant agreement 610711 (CACTOS), the Swedish Research Council (VR) under contract number C0590801 for the project Cloud Control, and the Swedish Government's strategic research program eSSENCE.

Acknowledgements

I, Mina Sedaghat, the author of this book, hereby confess that the acknowledgement section is written with much more love and enthusiasm than any other produced content in this thesis. I also confess that the people and locations named in this thesis are all fictional and any resemblance in name or reputation is a complete coincidence. These people are too good to be true.

First and foremost, I want to thank my advisor, Prof. Erik Elmroth. I, stronger than ever, believe on what I wrote in my licentiate thesis, ”Erik, you are the Best!”. Saying more will devalue my deep gratitude to you as my advisor, my mentor and a great friend[1]. Thanks for being selfless and supportive, frank, thorough and hard to please. Thanks for being kind in excess.

I also want to thank my co-advisor, Francisco Hernández, for his courage to make me an independent researcher, for his insightful comments on my papers and for giving me the freedom to explore. I am grateful to my co-authors, outside the department, John Wilkes, Sara De Luna, Oleg Seleznjev, and Sarunas Girdzijauskas for our fruitful discussions, their time and encouragements (some even despite the time differences).

To my official office-mate Gonzalo and my non-official office-mate Jakub. For being patient during my impulsive singing practices, for the ”soaked” moments, and for our heated debates on politics, science, breast cancer, and cake/cookie recipes. To my (ex)colleagues in the group, the inseparable couple (Pe**tt**er & Viali), Cristian, Johan, Ahmed, Ewnetu, Luis, Lars, Peter, P-O, Daniel, Abel, Muyi, Amardeep, Selome, and Ali. For being fun, nerdy and special to work with. For all the great moments that we have had together. I love you all and I am sure that with our true commitment, we can solve TV licence issues throughout the country. Special thanks to Lottas Krog for comforting us with food and drinks in all our group gatherings during the past few years.

To my friends at the department, Andrii, Eddie, Carl Christian and Helena, for teaching me how to take an average, how to think in math, how to love snakes, and how to dance freely. Thanks to the administrative staff for being patient and helpful through the administrative burden. To Mats, Bertil and Stric for being fun and generous in sharing the salt shaker.

To Mom and Dad, who I love the most...
To Mahbod, my role model...
To Shaghayegh, for being who she is...
and last but not least, to Ehsan, for his presence ...

[1] I know that you do not consider non-skiers as your friends but it is worth the shot.

Contents

Chapter 1

Introduction

The prospect of obtaining on-demand unlimited access to a wide variety of services and applications makes cloud computing an appealing paradigm for many users and organizations. Cloud computing offers its users the possibility to rent and customize their computing environments, immediately when they need them, to obtain a desired experience. As cloud computing has evolved, it has become simpler to plan, build and launch an IT platform targeting a large number of users. However, this increase in users' comfort has been accompanied by a steady increase in the challenge and complexity facing cloud operators as they strive to offer services that scale and satisfy expectations at several different levels.

Infrastructure is the main building block of cloud computing. Infrastructure Providers (IP) offer a pool of different kinds of hardware resources, including processing, storage, and network capacity, in a virtualized environment. These resources are usually pulled from several servers that may be distributed across multiple data centers, spread over different continents, in a seamless and transparent manner. Infrastructure providers also offer automated solutions for easy management of these resources, such as dynamic scaling, security, maintenance, backup and resiliency planning. These resources and services are available upon request and can be accessed from any location provided that an internet connection is available.

The quality of these services is usually agreed between the provider and the user via a Service Level Agreement (SLA). To guarantee a service level, the IP must determine the necessary quantity and type of resources to allocate to each customer, and how these resources should be placed. This is a critical and complex decision because the SLA should be maintained and guaranteed regardless of workload fluctuations, power outages, hardware failures, and any of the other unpredictable adverse events that may occur within the data center. Such decisions require rigorous monitoring, workload behavior prediction, smart strategies to achieve failure tolerance, capacity planning, admission control, and careful mappings of the demand to the available resources. The problem is made more complex by the fact that the underlying infrastructure should support numerous applications with heterogeneous workloads, from short-lived batch jobs to performance-sensitive web services, which are to run on an assembly of commodity hardware with heterogeneous resource types [5].

The complexity of the problem and the dynamic nature of cloud workloads oblige cloud providers to invest heavily in automating their processes and to adopt smarter, more optimized solutions. This has led to the development of autonomic solutions, which automatically monitor the system's state, predict and identify necessary changes in the system's configuration, and adaptively react to these changes by initiating appropriate reconfigurations [14]. The automation of this whole process is commonly referred to as autonomic resource management, and is the main focus of this thesis.

Chapter 2

Resource Scheduling and Management

Resource management is the process of efficiently planning and deploying resources to meet users' demands. Resource management middleware resides over the hardware layer and provides an abstract pool of hardware resources, using virtualization technology. A key function of a resource manager is to hide the complexities of resource allocation and failure handling from its users, allowing them to focus on application development [34]. Using virtualization, the resource manager partitions the resources into sets of resource slots known as Virtual Machines (VM), which can be assigned to different batch or service jobs. A job is a collection of one or more tasks, running on the cloud resources and can either refer to a user interactive service, known as *service job*, or a batch process, known as *batch job* [23]. A data center's operational costs, energy consumption and flexibility are all highly dependent on how resources are partitioned and mapped to jobs' demand.

To achieve effective resource management, there are a set of design goals that should be met:

Cost efficiency: Large scale compute clusters are expensive in both financial and environmental terms, and should therefore be utilized efficiently [23]. This means that the resource manager should improve resource and power utilization, and maintain acceptable levels of both quantities even in the face of variable utilization rates and demand. The resource manager should also avoid downtime and SLA penalties because they are both costly and harmful to the reputation of the enterprise.

Energy efficiency: Energy efficiency and reducing negative environmental impacts are also critical [15]. In a typical data center, about half of the power is consumed by the hardware resources, with the rest going mostly to cooling [2]. Effective resource management can dramatically reduce the data center's power consumption and environmental footprint. Importantly, increasing energy efficiency reduces the data center's electricity consumption and thus its operational costs.

Reliability: Reliability is the ability to perform and operate as expected over a period of time. It is a critical variable in a data center operation and must be achieved even if it requires sacrificing efficiency. Hardware failures, service interruptions, and downtimes are very expensive and should be avoided [35]. Therefore, the resource manager should able to pro-actively plan for failures and minimize downtime.

Scalability: Steady and ongoing increases in the number of running applications, the size of data centers, and workload diversity mean that resource managers must scale to hundreds of thousands of machines and support millions of task scheduling decisions per second.

2.1 Challenges

To meet the above-mentioned objectives, a set of challenges should be addressed:

2.1.1 Admission control

When a job arrives at the system, the first task is to decide whether to accept and accommodate it or not. This is a critical decision, specially when the amount of work exceeds the amount of available resources [23]. The job should be accepted if the data center has sufficient resources and the job has sufficient quota to spend. In this context, the quota is the maximum amount of resource that can be used by the job.

Admission control would be simple if resource utilization was constant for all jobs running in the system. However, capacity planning and decisions about job admission are complicated by fluctuations in demand, uncertainties due to the unpredictable arrival and termination of the jobs, and the risk of contentions on different resource levels [32]. Inefficient admission control can lead to underutilization of resources and revenue loss, or to frequent resource contention and degradations in the performance. It is therefore important to smartly plan the admission process to achieve efficient utilization without sacrificing performance.

2.1.2 Resource scheduling

For efficient operation, virtualization is used to divide physical machines into multiple logical partitions. Each job is assigned to a partition of a resource to satisfy its capacity demands. However, how resources are partitioned and how they are assigned, highly impact the utilization efficiency. It is also important to be aware that a resource is a multi-dimensional entity (for example, a single physical machine has compute, memory, network, and IO capabilities), and statically partitioning it over a single dimension (e.g. only compute) will lead to inefficient utilization with respect to other dimensions (e.g. memory or network). This forces us to formulate the scheduling problem as a multidimensional bin-packing problem, which is a challenging combinatorial NP-hard problem.

The scheduling problem becomes more complicated still because different jobs and workloads have different SLA requirements, and may require different ways of bin-packing tasks together. For example, low priority batch jobs or housekeeping tasks (such as file-system cleanups or logging services) can be packed/scheduled on highly loaded machines to backfill the server and increase utilization. However, it is not reasonable to pack a latency-sensitive web application or critical monitoring jobs on such a server because the high load would cause resource contention and affect the service's performance.

Furthermore, different workloads have different tolerances for scheduling delays. It does not make sense to wait for minutes for the scheduler's decision on a task that only runs for few seconds. Such workloads are exemplified by data analytic tasks, which require quick scheduling and have low tolerance for throughput bottlenecks [21]. However, it is worth spending time to carefully schedule a long running performance sensitive web-service because scheduling is generally a one-time decision and such services cannot be easily re-scheduled or migrated.

Changes in cluster dynamics are inevitable in cloud data centers. The scheduler should be able to cope with these changes and perform corrective actions to fix any sub-optimal allocations that arise [3]. Typical corrective actions include live migrations, the implementation of backfilling strategies, and preemption. However, to avoid incurring avoidable costs or degrading performance, all of these corrective actions should be planned carefully.

Data locality is another key factor when scheduling resources. Forcing servers to perform a costly fetch of required information across the network causes large overheads and leads to straggler effects [38]. Therefore, it makes more sense to schedule tasks close to the location of the data. In other words, the scheduler should ensure data locality to minimize network load and task completion times, both of which may be adversely effected by long data transfer times.

Finally, the scheduler should support placement constraints. As reported by [30], approximately 50% of production jobs have constraints on machine properties. These constraints can be soft constraints that describe the job's preferences, hard constraints that specify particular requirements (e.g. the server should have a GPU), or complex affinity anti-affinity constraints [16] that are defined in relation to the scheduling of other jobs (e.g. two database servers should not be located on the same server). Having different types of constraints for large portion of jobs necessitates the support for constraint satisfaction while scheduling.

2.1.3 Server consolidation

Server consolidation is a management strategy for increasing utilization and reducing the total number of active servers. The concept was originally developed to reduce server sprawl situations in which multiple servers are under-utilized and the total amount of work to be done could be performed with fewer resources. The under-utilization may be due to resource defragmentation or over-estimation of required resources by the service owners [31]. The resulting waste of power or cooling within a cluster can be mitigated by running multiple jobs on a single machine, reducing the number of servers required. This process is often referred to as *server consolidation*.

While consolidation substantially increases utilization, it also complicates the scheduling problem. The first problem arises when colocated jobs compete over shared resources in highly loaded servers. This competition causes resource contention and interference between jobs, and hence leads to performance degradation. While virtualization offers some level of isolation for CPU and memory, interferences affecting cache, I/O channels, network links and power are unavoidable [17]. Therefore, it is important to consider the trade offs between increasing utilization and minimizing resource contention when planning a consolidation.

Colocating different jobs on a single machine also complicates identifying the causes of performance misbehavior [5]. It becomes more complicated to ensure performance and reliability as there are more interacting threads and processes and one misbehaving thread can disrupt the performance of other processes and jobs.

It is also common to over-commit resources when planning a consolidation. Overcommitment enables higher consolidation ratios but also increases the risk of resource contention and introduces hidden costs of corrective actions if the consolidation is not carefully planned. Live migration and job preemption are two examples of corrective actions that are both complex and expensive. Live migration reduces the service's performance during the migration period, increasing its bandwidth consumption and requiring costly reconfigurations. Job preemption increases both completion times and the load on the scheduler, which is obliged to repeat the scheduling process for the preempted job. Good consolidation planning must therefore include an assessment of the risks of resource contention and interferences, associated with varying workload behavior. In addition, the costs of re-configuration and possible performance penalties should be considered in relation to the possible benefits of consolidation. Finally, if an overload occurs despite this careful planning, the scheduler should carefully come up with an efficient corrective plan that has minimal costs and performance impacts.

Another risk of consolidation is the potential for resource stranding. Resource stranding is the problem of underutilization on one dimension of a multi-dimensional resource while other dimensions are fully utilized. It can occur as a consequence of co-locating jobs with similar resource requirements on the same machine (e.g. multiple compute intensive or memory intensive jobs) such that the one resource is saturated while other resource dimensions are unused and thus wasted. Resource stranding reduces the overall utilization of resources over the data center and should be carefully avoided when planning a consolidation.

2.1.4 Workload analysis and prioritization

The challenges of resource management and scheduling arise from the fact that heterogeneous workloads are now able to utilize shared resources in cloud data centers.

Therefore, it has become essential for resource managers to understand and learn different workload behaviors in order to correctly plan allocations. Understanding workload patterns will help resource managers to become aware of potential interferences among workloads and to plan accordingly. The negative impacts of resource stranding can also be mitigated if the resource access patterns are well understood. Finally, workload prioritization can increase resource utilization, when services with different availability requirements can share nodes without impacting each others performance [5]. It is thus clear that intelligent resource management requires a deep understanding of workloads and their behaviors.

2.1.5 Fault tolerance

Data centers are always vulnerable to different types of faults, including hardware, network, and power failures. This vulnerability stems from the use of commodity hardware and the complexity of the underlying infrastructure in the data center. The failure-prone nature of data centers, together with users' increasingly high expectations of availability and reliability, mean that resource management solutions must be highly fault-tolerant. Resource manager should be capable of planning ahead for the failures and ensure that the system performs as expected, even in the presence of unexpected failures [36].

It should be noted that it is impossible to entirely eliminate failures. However, it is possible to mask them in various ways (for example by replication or fault-aware placement) to deliver a given level of job reliability [9]. The delivery of job reliability cannot be seen as something that is independent of the scheduling problem because a job's reliability is strongly dependent on the initial placement of tasks over different fault domains. To achieve the reliability, each scheduling and allocation decision should take into account both the probabilities of failure for different components, and the impact of these failures on service reliability.

2.1.6 Capacity auto-scaling

Auto-scaling is a feature offered by cloud providers that allows users to add and remove resources on demand, depending on their actual usage. This makes it possible to avoid over-provisioning and under-utilization by provisioning resources just as required. Resources can be scaled either horizontally or vertically. *Horizontal* scaling involves letting users increase or reduce the *number* of VMs allocated to their services, while *vertical* scaling involves letting them change the *configuration* and *volume* of the allocated resources (e.g. by adding a core or increasing the allocated memory) as their demand changes.

However, automating the dynamic scaling of resources is complex because any such solution should be scalable, robust, adaptive, and capable of making rapid decisions [1]. Furthermore, the scaling decisions should improve overall availability while reducing the data center's operating costs. The two key questions that an effective auto-scaler must answer are *when* should a given application's resources be scaled, and by *how much*? A deep understanding of the application's behavior and workloads is very beneficial when attempting to answer these questions.

2.2 Centralization vs. Decentralization

Resource managers should simultaneously support multiple objectives, some of which may conflict with each other. To achieve some objectives it is necessary to make high quality, system-wide decisions. In these cases, the management layer benefits from having comprehensive information on the state of all the resources under its management (including their consumption) as this makes it possible to evaluate all possible options [8]. To this end, the resource manager can be designed as a *single centralized controller* that monitors the whole system and reacts to the system state changes accordingly. Because a centralized design has only one decision maker, the resulting decisions are more likely to be consistent, efficient and aligned with high level objectives of the data center than would be the case for a decentralized design.

However, modern workloads pose new challenges and requirements that make centralization of control less efficient:

- **The need for high throughput decision-making**: Typical cloud workloads consist of a combination of many short batch jobs and a smaller number of longer-running service jobs. Short batch jobs such as a Spark [37] queries or Dremel [19] jobs are usually highly latency sensitive, as they only run for hundreds of milliseconds [21]. It clearly does not make sense to spend a few minutes to schedule such a job. Jobs of this class can represent up to 80% of all arrivals at a data center [23, 22, 13, 4] and therefore require a resource manager that scales to high arrival rates, supports high throughput, and makes decisions with minimum latency. When seeking to support high throughput, it is not sensible to make decisions in a serialized fashion as is typically done in centralized controllers [8, 7]. It is instead preferable to parallelize the decision-making process and divide the tasks over multiple decision makers such that one has a group of decision makers working concurrently to provide a solution. However, parallelization and faster decision-making are incompatible with a comprehensive overview of the system, so the quality of individual decisions will generally be reduced.

- **Workload heterogeneity**: The diversity of cloud workloads and applications has led to the development of a multitude of cluster computing frameworks [10], including Map-Reduce [6], Spark [37], Dryad [12], Pregel [18], and Pig [20]. These frameworks do not necessarily share a single scheduling logic, so a single resource manager cannot be optimal for all of them. Therefore, the resource manager should be flexible to support different scheduling logics. This can be done by dividing the functionalities among different independent schedulers, each handling a specific computing framework or a scheduling logic. In such scenarios, the resource manager shares the underlying resources among different schedulers, each of which performs its own scheduling. The viability and attractiveness of this approach are demonstrated by the growing number of decentralized schedulers, which include Omega [23], Mesos [11], YARN [33], Apollo [3] and Hawk [7].

- **Reliability and fault tolerance**: Decentralization of the management layer can also improve the system's reliability and fault tolerance. In a decentralized design, none of the controllers rely on a centralized state so the failure of one will not affect the operation of the others [21]. This is obviously not the case for a centralized scheduler because the single controller represents a single point of failure for the whole system.

- **Scalability**: The extent to which increasing scalability can be seen as a major motivation for decentralization is somewhat controversial. While some people believe that centralized solutions are still capable of responding at the current scale, others argue that centralized solutions are not sustainable given the increasing demand for cloud services. In general, although scalability is not necessarily the main motive for decentralization, it is effectively obtained as a "bonus" of a decentralized design.

Chapter 3

Contributions

Data center architecture and management have undergone a dramatic transition over the years, from single-purpose traditional hosting infrastructures to the more complex systems that are currently known as autonomic data centers. This evolution has been driven by a major increase in market demand with respect to both volume and agility expectations. These increases in demand, complexity, and scale have clearly revealed the need for abstraction and automation of the data center's processes.

To automate a complex system such as a cloud data center, it is helpful to break it down into a number of subsystems and manage each of them individually. This makes the individual subsystems more comprehensible and allows each one to deal exclusively with a specific sub-problem, collection of servers, or objective. However, the isolated management of individual subsystems is often less effective than centralized management of the system as a whole, and the collective outcome of individual managing units' decisions will not necessarily be optimal with respect to a data center's high level management objectives.

This thesis investigates the problem of resource management and scheduling of resources in a cloud data center. We also discuss modeling and coordinating the autonomic elements to achieve a high level management objective. It begins by exploring ways of defining resource management in terms of a collection of autonomic elements, and of integrating such elements into a single management framework that works toward an overall management objective. This is followed by the introduction of a decentralized resource management framework, which is a conceptual integration of independent autonomic elements in a way that enables them to collectively satisfy the data center's objectives. In the proposed framework, the high-level objectives are achieved as a consequence of the emergent behavior of the elements. Each element is responsible for different levels of decision making, control or actuation, at either the resource- or job level. The autonomic elements use local monitoring data to make local decisions, interact and cooperate with other agents to establish a global view of the system, and ultimately induce system-wide responses. A major difficulty in designing a decentralized system is the lack of a consistent global overview of the system, which contributes to the quality of decision-making. It is also challenging to achieve a given global response on the basis of a collection of local actions.

3.1 A decentralized resource management framework

We introduce a conceptual decentralized resource management framework in which the autonomic elements are structured as peers in a Peer to Peer (P2P) structure. The framework is based on a two-level resource management mechanism. On the first level, we propose a distributed scheduler and we study how such a scheduler decides which resources to allocate to the jobs, how to co-locate jobs to avoid interferences, and how to plan corrective actions. On the second level, we study the functionalities of a job level scheduler. We introduce algorithms on how to determine the amount and the type of the resources required to meet the demand of each job, how to schedule the job to guarantee a certain reliability, and how to scale it cost-efficiently. The framework features two main classes of autonomic elements:

1. **Node agents**: *Node agents* are schedulers that are responsible for infrastructure level resource management, i.e. the allocation and consolidation of jobs over the physical machines in the data center. Each *node agent* uses a sampling service to dynamically define its neighborhood and interact with other node agents within that neighborhood. The *node agent* collects information about available resources within its neighborhood and uses this local view to schedule a job locally or pass it on to another neighboring node agent for more efficient scheduling. This design means the system benefits from a high degree of concurrency and decentralization of control with no central bottleneck. Moreover, the *node agent* can adapt its behavior to satisfy different scheduling preferences relating to, for example, speed vs quality. To this end, the *node agents* adjust their sampling size and interactions based on the job's requirements and preferences.

   In addition to control decisions, the *node agents* can initiate and plan corrective actions if their associated servers are in a sub-optimal state (i.e. over- or underloaded). The *node agent* takes different factors into account when deciding on allocations and consolidations, and when drawing up a migration plan. Among the factors considered are the utilization, the scheduling latency, the risk of overload, and resource contentions. Factors considered when planning a migration or other corrective action to deal with over- or underload include the costs of reconfiguration, the volume of data transfer required for a given configuration, topological constraints, and the workload's sensitivity to migration. The *node agent* can account for placement constraints when deciding on allocations despite having no centralized controller.

   The *node agent* also acts as an admission controller if it cannot find sufficient resources to accommodate the request. Figure 1 shows an overall view of a

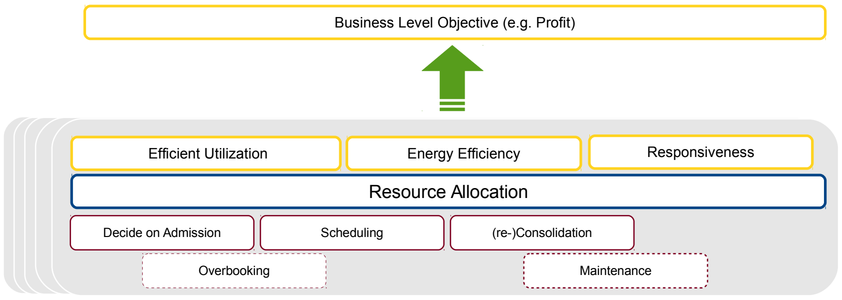


Figure 1: A node agent and its goals, domains of action, and controls

node agent's objectives and actions. More details are presented in papers [26, 28, 27].

2. **Job agents**: *Job agents* are simply job schedulers. Each *job agent* is responsible for satisfying the expectations of a single application, such as scaling the resources, repacking them to make them more cost-optimal [25], and ensuring the job reliability [29]. They are also responsible for communicating the job's scheduling preferences such as its placement constraints, application type, delay sensitivity, and resource requirements to the *node agents*. This two-level architecture with independent *job agents* for each application increases scheduling flexibility because each *Job agent* can have its own scheduling logic and preferences. Figure 2 presents an overall view of a *job agent's* objectives and actions. More details are presented in papers [25, 29].

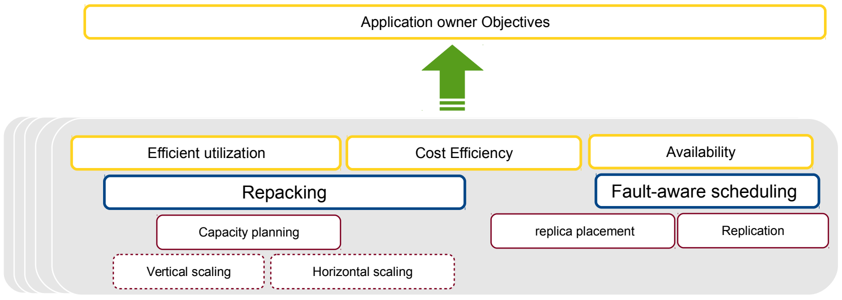


Figure 2: A job agent and its goals, domains of action, and controls

3.2 Summary of Papers

3.2.1 Paper I

Paper I [24] provides a bird's eye view of the resource management problem in a cloud data center. It reviews the challenges of cloud data center automation and recent attempts to address these challenges, analyzing them in a top-down fashion. As shown in Figure 3, from a top-down perspective the management process can be divided into a number of low-level controllers with distinct responsibilities. Each controller serves a specific purpose such as admission, auto-scaling, scheduling and allocation, or fault management.

The paper also argues that there is an essential need for coordination between these controllers. This claim is based on the fact that even though each controller has a distinct set of responsibilities, they are still far from independent because the decisions of one can have profound impacts on the behavior of others. Each controller is complex enough to make the system's implicit dependencies hard to capture. For example, increasing the consolidation level may allow the admission controller to accept more services. However, an aggressive admission policy that does not adequately consider the constraints on consolidation may increase the interference between the deployed jobs and thus reduce the benefits of consolidation. Therefore, it is essential to understand the dependencies among the controllers to ensure the optimality of the obtained solutions with respect to high level objectives.

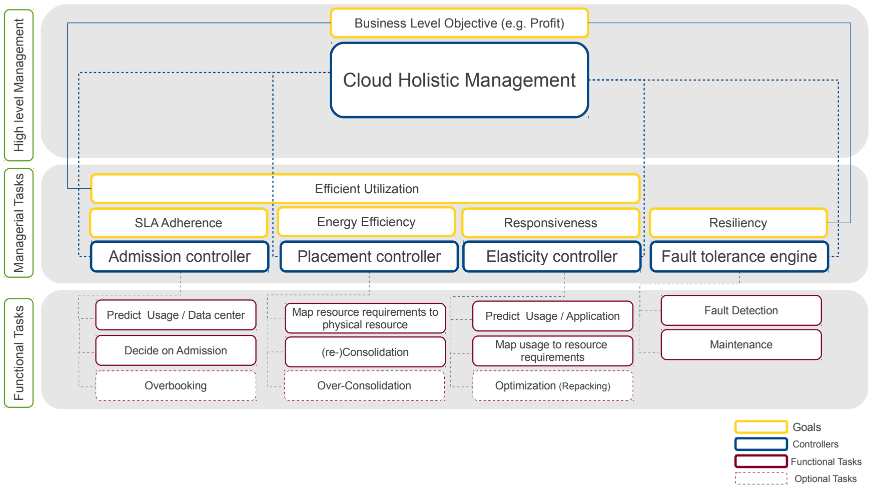


Figure 3: Resource Management (Holistic Approach)

Finally, the paper formulates the problem of controller coordination in a cloud data center, lists the major challenges to be overcome in this area, and proposes a business-oriented governance model to coordinate controllers' behavior in the presence of conflicting goals and to tune the system so that its individual elements work jointly toward a high level objective.

3.2.2 Paper II

Paper II [26] lays the foundations for the development of a P2P resource management framework for cloud data centers. The paper's objective is to develop a resource manager that is scalable with respect to the number of servers and incoming VM requests. The paper highlights the advantages of decentralized resource management frameworks and the challenges encountered in this design. It argues that the main motivations for creating such systems are scalability, flexibility and parallelization of control, while the challenges that need to be addressed include the lack of a global view and the consequent impact on the quality of the obtained solutions.

The paper proposes a decentralized architecture for a resource management framework with a P2P structure. Each peer in the framework is responsible for making local decisions based on its local information. Global objectives are achieved as an emergent outcome of these local interactions and optimizations.

The proposed framework has a 3 layer structure. The first layer is an overlay network, which specifies the logical interconnections between peers. The structural properties of the overlay affect the efficiency of information propagation within the system, which is essential to compensate for the lack of a global view. The overlay network is designed as a scale-free network.

The second layer consists of an agent community that interacts in a goal-oriented P2P fashion. Each agent (peer) is an autonomous resource scheduler that acts on behalf of a physical server. Finally, the third layer is a collection of agents that act as job schedulers.

As part of the framework, the paper focuses on the problem of resource allocation and VM consolidation, which is discussed at length in the paper. The allocation problem is modeled as an optimization problem, with the goal of maximizing resource utilization. A local search heuristic is proposed as a solution. The paper argues that the efficiency of allocations in a fully distributed structure is highly dependent on the efficiency of information propagation and the distribution efficiency. Effective information propagation and discovery reduces the computational time, while an efficient allocation distribution reduces defragmentation and thus increases utilization.

The system's performance is evaluated in terms of data center utilization, server utilization, number of hops (used as a measure of computational time), rejection ratios, and profit. The scalability of the system is also studied by evaluating its performance in simulations with different numbers of servers. The results show that the system can scale up to 5000 servers with 9000 placement requests arriving during one hour of simulation. Finally, we present a re-consolidation process to continuously optimize sub-optimal allocations, occurred as the result of workload changes. In our experiments, reconsolidation increased node utilization by up to 10%.

3.2.3 Paper III

Paper III extends the framework proposed in Paper II, focusing on the placement of 2-dimensional VMs and the re-consolidation of multi-dimensional resources. The paper tackles the problem of resource stranding in data centers and introduces a gossip protocol to mitigate its negative impact. Within the proposed protocol, placement decisions are made continuously between random pairs of cooperative agents that are trying to improve a common value. The common value is defined as the total imbalance in utilization of each pair at the time of decision making, and the goal is to reduce this imbalance by redistributing the VMs between the two agents. The cooperative approach prevents undesirable bounces of VMs and ensures a stable state, which is essential to avoid redundant migrations.

Moreover, in the extended framework, the logical overlay is dynamically built and maintained by a peer sampling service. This stands in contrast to the system considered in paper II, in which the overlay is a static scale-free network.

The feasibility, scalability and performance of the proposed framework and its consolidation algorithm were evaluated by simulating a data center that has 100,000 servers and receives 200,000 VM requests during the simulation time. The VMs are assumed to be capable of running a combination of batch jobs and stateless interactive services. The requests' types are uniformly distributed among memory-optimized, compute-optimized, and general purpose VM types. The results show that the proposed decentralized approach is feasible and scalable over the studied range, and can produce placement decisions within a short computation time. Specifically, the protocol converges in less than 7 cycles in all studied cases. The observations also indicate that a balanced utilization of a two dimensional resource in a mixed workload results in more efficient utilization of resources in both dimensions. This reduces the incidence of rejection of future VM requests and accelerates the resolution of overloaded servers. Overall, use of the framework reduced the incidence of rejection by 28%, increased the rate of offload by 12.6% (for a greater number of overloaded servers than was possible with the previous method), and reduced power consumption by 3.2%, at the expense of a 25% increase in migration incidence.

3.2.4 Paper IV

Paper IV [27] investigates the placement and re-consolidation problem, focusing on the costs of different corrective actions such as backfilling and migration. The aim is to identify a way of planning and maintaining durable allocations to reduce the cost of corrective actions for data centers with dynamic and heterogeneous workloads. The proposed scheduler assigns resources to VMs and maintains their packing efficiency while taking into account migration costs, topological constraints, and the risk of resource contention, as well as the variability of the background load.

We extend the sampling protocol used in Paper III to support topological awareness when planning for migration. The new sampling protocol provides a neighbor list that is based on nodes' physical proximity in the data center architecture, in addition to the each peer's timestamp. A server has a higher probability of being returned as a sample if it is in the same server group as the main peer. The P2P overlay creates logical dynamic connectivity (dynamic partitioning) within a large pool of resources and reduces the negative impacts of static partitioning, which can cause low utilization. However, considering the physical proximity of the neighbors when building the logical overlay reduces the costs of network transit and reconfigurations.

To perform consolidation, node agents initiate a resource discovery process in which they rank their neighbors on the basis of their proximity, the risks of resource contention and load variations, and efficiency of allocation both in terms of utilization and proportionality of consumption over different resource dimensions. The new migration planning heuristic also accounts for possible data transfer costs when planning a migration. At the end, the node agent selects the best candidate among the neighbors, found within the acceptable discovery time.

The proposed heuristic is evaluated by simulating a data center with over 65000 servers that are interconnected in a multi-rooted tree topology, for a simulation period of 24 hours. The efficiency and durability of consolidation, data transfer efficiency, number of migrations, and their localization were analyzed by measuring 20 different performance metrics. We compared our risk- and topology-aware heuristic, which is referred to as the Reconsolidating PlaceMent scheduler (*RPM*), to the commonly used multi-dimensional First Fit Decreasing (*FFD-sum*) bin-packing heuristic. The results show that *RPM* reduces the cost of maintaining the desired packing efficiency. By accounting for the risks of variability and load contention, *RPM* reduces the number of sub-optimal state triggers by up to 60% compared to *FFD-sum* and thus reduces the number of migrations required to resolve sub-optimal states. Finally, the paper discusses the impact of individual variables considered within the proposed heuristic function, and identifies its key performance trade-offs.

3.2.5 Paper V

Paper V [25] addresses an application-level concern: how to cost-effectively scale resources in a cloud data center. The paper introduces a *repacking* approach to automatically manage the capacity acquired by an elastic application in a cost-effective and on-demand manner.

The paper studies the trade-offs between two commonly used scaling approaches, *horizontal scaling* and *vertical scaling. Horizontal scaling* can quickly adapt an application's resource set by adding extra capacity, without any need for re-configuration of the currently deployed VM set. However, the resulting resource set can become far from optimal for the aggregated capacity over time. *Vertical scaling* can maintain the optimality of the resource set, at the cost of expensive and time-consuming re-configurations for each change in demand. The paper investigates how combining the benefits of both scaling strategies can improve the cost-efficiency of an elastic application.

A cost-benefit analysis is presented to determine the trade-off between *horizontal* and *vertical* scaling. This analysis is used to support the design of a repacking controller that evaluates the performance of the current configuration as well as the cost and durability of proposed reconfigurations when deciding when and how a repacking should be performed.

The proposed repacking controller is compared to different auto-scaling strategies. A sensitivity analysis is carried out to study the impacts of different parameters and configuration settings on *repacking* decisions. The analysis shows that combining the benefits of *horizontal* and *vertical* scaling, and effectively replacing the non-optimal resource set can reduce an application's total cost of resource utilization by up to 60% over its lifetime.

This functionality is studied as part of the *job agent* functionalities and can be embedded in the resource management framework.

3.2.6 Paper VI

Paper VI [29] introduces a scheduling algorithm that considers probabilistic failures when scheduling a job. The work presents a reliability model for a job running in a virtual data center, with the possibility of stochastic and correlated failures and different failure characteristics. Drawing on the model's properties, a scheduling algorithm is proposed to ensure job reliability in the presence of correlated failures. The goal is to minimize the number of concurrent failures due to a single failure event, and to maintain the desired reliability with the minimum number of extra tasks.

The proposed the scheduling algorithm approximates a minimum number of required tasks and a placement to guarantee a desired job reliability. We study the efficiency of our algorithm using an analytical approach and by simulating a cluster with different failure sources and reliabilities. The results show that the algorithm can effectively approximate the minimum number of extra tasks required to guarantee the job's reliability. This functionality is also studied as part of the *job agent* functionalities and can be embedded in the resource management framework.

3.3 Future outlook

Decentralized resource management is shown to be efficient, scalable, and able to cope well with highly dynamic systems. Consequently, there is great scope for its exploitation in a wide range of environments including geo-distributed data centers, telecom clouds, and Internet of Things (IoT) environments.

This thesis presents a decentralized resource management framework and a set of algorithms and controllers for managing resources within a cloud data center. However, the ideas described herein are also applicable to environments where the resources are actually distributed. In such environments, the cost of maintaining a global overview of the system's state is substantial because it would necessitate sending large volumes of data over the network to a centralized controller. This would also make the decision delay non-negligible, because the network transmission delay would be added to the processing time. Local decision-making based on local knowledge is very advantageous in such environments because of the potential to minimize unnecessary communication and increase robustness.

Moreover, the number of resources in contexts such as an IoT environment or a telecom cloud may be much greater than that in a single data center. This brings scalability concerns to the forefront and raises important questions about the suitability of centralized controllers in such settings.

To effectively apply the techniques introduced in this thesis in such environments, we need to better understand their dynamics, investigate which global objectives can be achieved using only local interactions, and optimize the framework and its interconnections on the basis of realistic assumptions and/or robust characterization data concerning the environments of interest and their global objectives.

Bibliography

[1] Ahmed Ali-Eldin, Maria Kihl, Johan Tordsson, and Erik Elmroth. Efficient provisioning of bursty scientific workloads on the cloud using adaptive elasticity control. In *Proceedings of the 3rd workshop on Scientific Cloud Computing Date*, pages 31–40. ACM, 2012.

[2] Luiz André Barroso and Urs Hölzle. The case for energy-proportional computing. *Computer*, (12):33–37, 2007.

[3] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. Apollo: scalable and coordinated scheduling for cloud-scale computing. In *Proceedings USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, 2014.

[4] Yanpei Chen, Sara Alspaugh, and Randy H Katz. Design insights for mapreduce from diverse production workloads. Technical report, DTIC Document, 2012.

[5] Walfredo Cirne and Eitan Frachtenberg. Web-scale job scheduling. In *Job Scheduling Strategies for Parallel Processing*, pages 1–15. Springer, 2013.

[6] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[7] Pamela Delgado, Florin Dinu, Anne-Marie Kermarrec, and Willy Zwaenepoel. Hawk: hybrid datacenter scheduling. In *2015 USENIX Annual Technical Conference*, pages 499–510. USENIX Association, 2015.

[8] Christina Delimitrou, Daniel Sanchez, and Christos Kozyrakis. Tarcil: Reconciling scheduling speed and quality in large, shared clusters. In *2015 ACM Symposium on Cloud Computing (SoCC 2015)*, 2015.

[9] Daniel Ford, François Labelle, Florentina I Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. Availability in globally distributed storage systems. In *OSDI*, pages 61–74, 2010.

[10] Benjamin Hindman, Andrew Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Scott Shenker, and Ion Stoica. Nexus: A common substrate for cluster computing. In *Workshop on Hot Topics in Cloud Computing*, 2009.

[11] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, volume 11, pages 22–22, 2011.

[12] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 59–72. ACM, 2007.

[13] Soila Kavulya, Jason Tan, Rajeev Gandhi, and Priya Narasimhan. An analysis of traces from a production mapreduce cluster. In *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, pages 94–103. IEEE, 2010.

[14] Jeffrey O Kephart and David M Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[15] Jonathan G Koomey. Estimating total power consumption by servers in the us and the world, 2007.

[16] Lars Larsson, Daniel Henriksson, and Erik Elmroth. Scheduling and monitoring of internally structured services in cloud federations. In *Computers and Communications (ISCC), 2011 IEEE Symposium on*, pages 173–178. IEEE, 2011.

[17] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: improving resource efficiency at scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 450–462. ACM, 2015.

[18] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the ACM international Conference on Management of data (SIGMOD)*, pages 135–146. ACM, 2010.

[19] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: interactive analysis of web-scale datasets. 3(1-2):330–339, 2010.

[20] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the ACM international conference on Management of data (SIGMOD)*, pages 1099–1110. ACM, 2008.

[21] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, (Sosp)*, pages 69–84. ACM, 2013.

[22] Charles Reiss, Alexey Tumanov, Gregory R Ganger, Randy H Katz, and Michael A Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing*. ACM, 2012.

[23] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 351–364. ACM, 2013.

[24] Mina Sedaghat, Francisco Hernández, and Erik Elmroth. Unifying cloud management: Towards overall governance of business level objectives. In *11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 591–597. IEEE, 2011.

[25] Mina Sedaghat, Francisco Hernandez-Rodriguez, and Erik Elmroth. A virtual machine re-packing approach to the horizontal vs. vertical elasticity trade-off for cloud autoscaling. In *Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference*. ACM, 2013.

[26] Mina Sedaghat, Francisco Hernández-Rodriguez, and Erik Elmroth. Autonomic resource allocation for cloud data centers: A peer to peer approach. In *2014 International Conference on Cloud and Autonomic Computing (ICCAC)*, pages 131–140. IEEE, 2014.

[27] Mina Sedaghat, Francisco Hernández-Rodriguez, and Erik Elmroth. Decentralized cloud datacenter reconsolidation through emergent and topology-aware behavior. *Future Generation Computer Systems*, 56:51–63, 2016.

[28] Mina Sedaghat, Francisco Hernández-Rodriguez, Erik Elmroth, and Sarunas Girdzijauskas. Divide the task, multiply the outcome: Cooperative VM consolidation. In *2014 IEEE 6th International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 300–305. IEEE, 2014.

[29] Mina Sedaghat, Eddie Wadbro, John Wilkes, Sara De Luna, Oleg Seleznjev, and Erik Elmroth. Die-hard: Reliable scheduling to survive correlated failures in a cloud datacenter. *Submitted for publication.*

[30] Bikash Sharma, Victor Chudnovsky, Joseph L Hellerstein, Rasekh Rifaat, and Chita R Das. Modeling and synthesizing task placement constraints in google compute clusters. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*. ACM, 2011.

[31] Luis Tomás and Johan Tordsson. Cloudy with a chance of load spikes: Admission control with fuzzy risk assessments. In *2013 IEEE/ACM 6th International Conference on Utility and Cloud Computing (UCC)*, pages 155–162. IEEE, 2013.

[32] Luis Tomás and Johan Tordsson. Improving cloud infrastructure utilization through overbooking. In *Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference*. ACM, 2013.

[33] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 2013.

[34] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*. ACM, 2015.

[35] Kashi Venkatesh Vishwanath and Nachiappan Nagappan. Characterizing cloud computing hardware reliability. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 193–204. ACM, 2010.

[36] Praveen Yalagandula, Suman Nath, Haifeng Yu, Phillip B Gibbons, and Srinivasan Seshan. Beyond availability: Towards a deeper understanding of machine failure characteristics in large distributed systems. In *WORLDS*, 2004.

[37] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, volume 10, page 10, 2010.

[38] Matei Zaharia, Andy Konwinski, Anthony D Joseph, Randy H Katz, and Ion Stoica. Improving mapreduce performance in heterogeneous environments. In *Operating Systems Design and Implementation, (OSDI)*, volume 8, pages 29–42, 2008.

Paper I

Unifying Cloud Management: Towards Overall Governance of Business Level Objectives

M. Sedaghat, F. Hernandez-Rodriguez, and E. Elmroth

In Proceedings of the 11th *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pp. 591-597, IEEE Computer Society, 2011.

Unifying Cloud Management: Towards Overall Governance of Business Level Objectives[*]

Mina Sedaghat[†], Francisco Hernandez-Rodriguez[†], Erik Elmroth[†]

Abstract

We address the challenge of providing unified cloud resource management towards an overall business level objective, given the multitude of managerial tasks to be performed and the complexity of any architecture to support them. Resource level management tasks include elasticity control, virtual machine and data placement, autonomous fault management, etc, which are intrinsically difficult problems since services normally have unknown lifetime and capacity demands that varies largely over time. To unify the management of these problems,(for optimization with respect to some higher level business level objective, like optimizing revenue while breaking no more than a certain percentage of service level agreements) becomes even more challenging as the resource level managerial challenges are far from independent. After providing the general problem formulation, we review recent approaches taken by the research community, including mainly general autonomic computing technology for large-scale environments and resource level management tools equipped with some business oriented or otherwise qualitative features. We propose and illustrate a policy-driven approach where a high-level management system monitors overall system and services behaviors and adjusts lower level policies (e.g., thresholds for admission control, elasticity control, server consolidation level, etc) for optimization towards the measurable business level objectives.

Keywords: Cloud governance; Autonomic computing; Policy-driven management.

[*]The paper has been re-typeset to match the thesis style.

[†]Dept. of Computing Science, Umeå University, SE-901 87 Umeå, Sweden, email: {mina, francisco, elmroth}@cs.umu.se

1 Introduction

Cloud management for optimizing a cloud provider's overall objectives is becoming increasingly more difficult following the rapidly increasing scale of resources and services to be managed and the intrinsic challenges of fundamental cloud management. Already individual fundamental cloud management issues such as admission and elasticity control, virtual machine (VM) and data placement, autonomous fault management, etc. are difficult since services normally have unknown lifetime and capacity demands that vary largely over time.

The overarching management goal of cloud infrastructure providers (IPs) is often referred to as the managerial Business Level Objective (BLO). For a commercial IP this BLO may, for example, be to manage resources so as to optimize revenue[1] while keeping customers sufficiently happy[2]. For a non-commercial IP, the BLO may, e.g., be expressed in terms of maximizing resource utilization while maintaining a specified level of fairness between users [1].

The overall management problem becomes the complex challenge of managing the individual tasks of admission and elasticity control, VM and data placement, fault management, etc, and to optimize the overall system behavior with respect to the BLO. An attractive way of dealing with the complexity is to separate functionalities into individual and to some degree autonomous components with clear separation of concerns, e.g., including separate components for admission control, elasticity control, VM placement, data placement, fault management, etc. However, these problems are not totally independent. For example, increasing the server consolidation level (number of VMs per physical host) or making the elasticity control more restrictive would allow for admission control to be less restrictive in accepting services. Hence, there is a need for some degree of coordination between such actions. In this work we call this coordination governance. This problem is further examined in Section 2.

Recent studies on this type of management issues for clouds are mainly focusing on either the general autonomic computing aspects supporting management of truly large scale environments or on how to enhance cloud resource level functionalities by adding business or qualitative features. Section 3 reviews several existing solutions on management in clouds with respect to business level concerns. The review is followed by a description of current challenges in designing a business oriented governance model, presented in Section 4.

[1]Revenue calculation can be arbitrary complicated but should at least take into account parameters such as income, operation and infrastructure costs and costs associated with violated service level agreements (SLAs).

[2]For resource management, a criterion such as keeping customers sufficiently happy may, e.g., be translated into constraints on maximum fractions of SLAs to violate.

Section 5 proposes a governance model that aim to direct and control a set of lower level management tools so that their concerted actions strive towards the overall BLO. Finally, Section 6 concludes the paper and outlines possible future research directions.

2 Cloud management

Current efforts in cloud resource and service management mainly focus on the most technical and fundamental aspects of resource provisioning [2]. Managerial expectations and business objectives, although important, are seldom considered. One such expectation is efficiency in resource utilization with respect to SLA constraints and QoS requirements. Under normal circumstances services are not using their maximum required capacity all at once. Hence, in order to efficiently make use of available resources, infrastructure providers aim to sell more capacity (with respect to services expected peak loads) than what is physically available, i.e., controlled over-provisioning[3]. To complicate matters, there are other challenges that need to be considered like the necessity to support elastic behaviors of services without breaking SLAs or SLA management mechanisms that prioritize services when not all SLAs can be met. Furthermore, mechanisms to avoid crashes and failures i.e., manage breakdowns and repair actions, are also needed. Key to handle these types of cloud management challenges is the ability to provide efficient and coordinated solutions to the following problems:

Admission Control (AC): IPs should be able to decide whether a service should be accepted or rejected, taking economic and other considerations into account.

Elasticity: IPs should be able to predict and manage the elastic behavior of services. Rapid changes in service demands should be handled by appropriate capacity adjustments, so that services can scale up or scale down easily.

Fault tolerance: IPs should be able to manage crashes and infrastructure failures. Unexpected resource failures should be managed through repair actions while prediction of possible upcoming crashes, in order to facilitate preventive actions, should also be considered.

[3]The term typically used is over provisioning. However, due to the possible negative connotation of the term, we prefer to explicitly state that the over provisioning is controlled, i.e., it is in line with the business goals of cloud providers.

Data placement: IPs should be able to place data at optimal resources taken into account, e.g., a range of performance aspects and resource availability.

VM placement: IPs should be able to place VMs on physical machines based on their SLAs and on an estimation of their future behaviors. Efficiency in resource utilization should also be taken into account.

SLA management: IPs should be able to enforce SLAs and manage violations based on priorities.

All of these challenges indicate specific problems that can be formulated and solved more or less independently. However, although each problem can be solved autonomously to satisfy its specific goals it should be acknowledged that the problems are intertwined. Table I, summarizes the aforementioned problems and introduces sample software components (from here on called low level managers) to address them. The table also shows the information required to tackle the problem as well as the outcomes expected from each low level manager. The outcomes can be a decision making or a specific action.

Ideally, all these problems should be solved in concert, striving towards a high level managerial objective, i.e. the resource management BLO. BLOs delineate a desired goal like maximizing profit without breaking more than a certain fraction of SLAs or (for a non-profit organization) to maximize resource utilization while maintaining fairness among users. Part of the challenge is that an optimal solution to one of the specific management problems may be in conflict with the concerted actions needed to optimize the BLO. For example, an AC can accept services as long as there are available resources, selecting between incoming services based on a first come first serve policy. An AC can also overbook resources without considering elastic behaviors of services, that are already running. In this case, even though the AC completely fulfills its own responsibilities, the outcome of the applied policy may not result in the revenue expected by the business manager. This may cause costly SLA violation penalties and reputation damages, both aspects affecting the BLOs. This is due to the AC's lack of interaction with other components like the SLA manager, or lack of information about economic factors like pricing.

Meanwhile, the elasticity controller can detect increases in service demand and it can decide to allocate extra resources to services. However, if this is done in isolation, there may not be enough resources available, if e.g., the placement engine also deploys new services. Deciding whether to increase or decrease the allocation of additional resources or to reject the deployment of the new service involves an evaluation of the impact of either action on the BLOs. As this overarching evaluation is in conflict with the goals of each manager, it requires a higher coordination. To summarize, the variety in business level objectives

Table 1: *Important cloud management challenges: Logical view. Notice that a real implementation may address several of these challenges in a single software component*

Low level manager	**Required information**	**Activities & Decisions**
Admission Controller	1.The available Capacity 2.Expected load from all services 2.SLA requirements per service 3.Historic workload 4.Price tariffs	1.Accept or Reject 2.Select a service from a queue
VM Placement Engine	1.The total amount of resources 2.The allocated resources 3.SLA including placement constraints 4.Data locality	1.Service placement 2.Service termination or cancellation 3.Service migration
Data Placement Engine	1.Current and predicted storage 2.End user locality including trends,network performance and characteristics	1.Data objects placement 2.Data objects migration
Elasticity Engine	1.Current load 2.Predicted future load 3.Historic load and trends 4.SLAs	1.Allocate additional resource 2.Release allocation
Fault Tolerance Controller	1.Resource status (Up, Failed, Probate) 2.List of crashed resources 3.List of crashed services 4.Possible repair action	1.Restart the service 2.Terminate the service 3.Postpone the restart 4.Preventive service migration
SLA Management Engine	1.SLA 2.Service state (e.g., the expected service completion time,service deadline) 3.Penalty per service	1.SLA prioritizing 2.SLA violation cost estimation

and strategies, and the need for scalable solutions to support diverse scenarios in a cloud environment illustrate the necessity of an autonomic higher level governance mechanism to handle these complexities.

The result of having a high level mechanism is an increase in the abstraction level of cloud management in which all high level decision making and BLO management is performed by a higher level governance manager [3–5]. This manager may adjust the behavior of low level managers with respect to the information collected from all resource level functionalities with the aim of satisfying the high level BLOs.

Advances in the operation of the low level managers are also needed. In the next section we present existent solutions on management approaches for enhancing the normal operation of low level managers with business level concerns, as well as other, more comprehensive, approaches for cloud management that relate to our work.

3 Cloud management survey

Different aspects of cloud management are partially studied in a number of research projects. In these projects, the problem is mainly addressed by adopting business concerns by a specific low level manager like VM placement engine or admission controller; formulating and solving optimization problems; or autonomic management.

3.1 Adoption of business concerns by low level managers

The low level managers presented below may all be considered traditional low level managers enhanced with features to optimize management with respect to some higher level management objective. These enhanced managers are designed to operate independently of other managers based on their own (self-centered) objectives, but the overall coordination in the system remains an unsolved issue.

Puschel et al. [6] focus on developing an admission control mechanism aimed to increase the revenue for an IP. The decision to accept or reject a service is made based on predefined policies for dynamic pricing and client classification. They develop a policy-based decision model by defining policies as heuristics when the environment is non-deterministic and there is not enough information about upcoming services. The policies are defined in terms of SLA's committed, previous workloads, utilization trends, and BLO's.

Perez et al. [7,8] address cloud scheduling and elasticity problems with respect to end users' satisfaction. Responsiveness and fair share are the adjustable high level objectives supported by the model. They propose a reinforcement learning approach for a resource allocation mechanism, because of its flexibility to adapt its decisions to the elastic behavior, QoS requirements and also the minimal knowledge requirement about the environment. The adaptability and optimal policy selection in this model has been achieved using an on-policy learning algorithm, called SARSA.

Moon et al., in a paper on cloud resource scheduling [9], argue that the success of clouds depends on QoS factors and cost management. Therefore, they propose an SLA aware resource scheduler that minimizes SLA penalty costs and optimizes profit based on a cost heuristic. The scheduler evaluates a number of jobs in the queue individually and picks the jobs in order of priority. The cost based scheduler uses a probability density function for computing the expected cost.

Optimizing fault tolerance and repair mechanisms for cost optimization and improvement of QoS and availability for an IP is addressed by Goldszmidt et al. [10]. They propose a framework for evaluation and optimization of policies governing automated repair services, adopting a two phase approach. The first phase estimates the effectiveness of each repair action by assessing historic data. The second phase uses the information processed in the first phase to refine the policy using machine learning techniques.

There are also a number of projects addressing BLOs in general, not focusing on a specific functionality but in supporting managerial objectives with respect to lower level constraints. SORMA [11] strives to fulfill the provider's ambitions on maximizing revenue and other business type objectives. In SORMA, the business management role is assigned to the EERM framework (Economically Enhanced Resource Manager) [12]. The EERM is a resource manager enhanced with business related features that addresses resource scheduling with respect to SLA management and admission control. The EERM makes use of a rule based policy manager to support adaptable policies that are formulated in Semantic Web Rule Language (SWRL). The overall aim of the EERM is to isolate SORMA's economic layers from the technical ones and orchestrate both economic and technical goals in order to achieve maximum economic profit (i.e. revenue) and resource utilization [6].

The Grid-Econ project [13,14] takes a broader view on business level concerns. Grid-Econ is essentially a resource broker built on a matching algorithm that considers quantity of resource units, the period of time over which the resource is required or available, the minimum selling price or the maximum buying price, and the expiration date of the request to buy or sell a resource [15]. Different types of risks and trust issues in grid markets to support non-commercial grid stakeholders such as insurance against resource failures, resource quality assurance, stable price offering and capacity planning are also addressed. However, all these qualitative objectives are defined as fixed goals in an auctioning strategy and are not flexible or easily extendable.

3.2 Optimization problems

Of interest are also optimization methods or algorithms that satisfy a business objective. Most of these algorithms work around optimization techniques and utility functions. Salehi and Buyya, [16] use a time and cost optimization for resource scheduling. They introduce two market oriented policies that consider user constraints such as time and budget. The produced schedules support the elastic behavior of services, load changes in services, and aim at satisfying deadlines by extending the computational capacity of local resources via hiring resources from cloud providers. Similarly, Silva et al. [17] propose a heuristic algorithm for dynamic resource allocation, with the same constraints on time and cost. The algorithm determines the optimal number of hosts for short lived tasks.

Paton et al. [18] demonstrate the use of utility functions in utility driven workload executions in clouds. Utility functions quantify and rank the relevance and desirability of each system state, i.e., the functions offer a common and consistent scale to compare the states and objectives. The states may represent response time, number of QoS goals met or income. The utility metrics discussed in this work are profit and response time. They also introduce an autonomic workload mapper responsible to assign tasks to available execution sites and to dynamically monitor and modify assignments during workload execution. The assignments are based on feedback on the overall progress of submitted requests.

3.3 Autonomic management

Autonomic management offers a more comprehensive approach to cloud management. The goal is to offer functionalities for self-configuration, self-management and self-healing. The idea is that by adding autonomic features to a system, the system would be able to adapt itself to real time configuration and changes. The general autonomic computing aspects support management of truly large-scale environments.

Unity [19] is an agent based autonomic data center with improved behaviors for a self-manageable computing system. It is structured as a set of autonomic components that can manage themselves and also interact autonomously with humans or other autonomic components. The addressed scenario is an elastic resource allocation between different clusters in a grid. The adaptation of the system to the high level policies is derived by a decomposition of the main policy represented as a utility function. An extension to the Unity Project, IBM Tivoli Intelligent Orchestrator (TIO) [20, 21] is an automated provisioning manager for Internet data centers. TIO is capable to automatically deploy and dynamically optimize resources in response to business objectives in heterogeneous environments [20], and it performs on-demand deployment by proactively sensing and responding to peaks in demand and allocating IT resources to the most important processes based on business policies [20]. Elasticity is supported with respect to load, bandwidth, CPU utilization, etc. through the use of a policy manager.

CERAS laboratory [3, 4, 22, 23] works on different aspects of business driven cloud architecture. They automate several activities like monitoring, analysis and prediction, planning and execution through a feedback loop controller that optimizes specific goals and constraints associated with the type of provider or service models, i.e., IaaS, PaaS, SaaS. The main objective that is studied is cost optimization which is a common interest between the three mentioned service models. Notably, [23] goes deeper by considering response time and mean throughput constraints, while taking into account resource contention.

FoSII (Foundation of Self-Governing ICT Infrastructure) [5] tries to develop a self manageable cloud environment that is autonomously complying with user requirements in terms of SLAs and also it achieves a level of flexibility. The main goal is avoidance of SLA violations and, to this end, advanced monitoring and knowledge management [24, 25] are required. FoSII uses a framework that senses infrastructure resource metrics and predicts the risk of SLA violations based on pre-defined thresholds and actual usage. In order to do this, the SLA parameters are refined and mapped into resource level metrics. These refinements are done based on pre-defined mapping rules defined through the use of Domain Specific Languages (DSLs).

4 Challenges of cloud governance

As described above, we view the cloud management problem as a set of management tasks that are addressed by independent managers. In this work, we propose to use cloud governance for coordinating the independent managers for overall optimization of the BLO. Developing the governance solution brings about several challenges classified in 4 main categories: BLO formulation; BLO interpretation to resource level objectives; policy enforcement on resource level engines and monitoring and feedback control.

4.1 BLO formulation

Current solutions in BLO formulation mostly express BLOs as fixed goals and formulate the management problem as a set of utility functions with multiple constraints. Cost, profit, and time optimizations are the most popular objectives employed [6, 9, 16, 18] but qualitative aspects such as fairness, responsiveness and utilization have also been considered [7, 8, 10]. However, fixed goals are unable to express the wide range of desired system behaviors that vary from one business to another [19]. For example, commercial providers focus on maximizing profit and reducing costs whereas fairness and utilization may be the most important features for an academic resource provider. Thus, a cloud management system should be adjustable and adaptable to a wide range of economic strategies.

4.2 BLO interpretation to resource level objectives

Policies map BLOs to system actions [26]. They translate BLOs into desired resource level behaviors. Several mechanisms have been employed to perform such translations. Emeakaroha et al. [24] employ a mapping table to translate SLAs into quantifiable resource level metrics, e.g. availability is mapped to a function of the time it takes for a failure to be repaired and the time system is available. Unity [19], Paton et al. [18] and Li et al. [27] make use of utility-function based policies that allow the human administrator to guide the operation of the system. The adaptation of the system to the high level policies is made through a decomposition of the utility function into sub-problems, and optimization of these sub-problems directs the resource level components toward the main objectives. In EERM [12], policies are formulated in the semantic web rule language (SWRL). All features of the EERM require the components to be able to communicate with the policy manager and base their decisions on the corresponding policies [12].

However, to define a relation between high level objectives and constraints to the system level metrics is not always possible, e.g., the number of policies may be unmanageable if the high level objectives vary frequently.

4.3 Policy enforcement on resource level engines

The overall goal of a governance engine is to direct and adjust the behavior of resource level managers. The capability of the system to adopt the policies is crucial for the operation of the governance manager since resource level managers are the actual enactors of the policies. To this end the governance manager must be able to exert action over the resource level managers. Furthermore, the governance manager adjusts the operation of the system by selecting appropriate policies to enforce, from the policy set of each of the resource level actuators, according to the operation of the system at any given moment and the expected BLOs.

4.4 Monitoring and feedback control

The impact of policy enforcement on the low level engines and on performance of the system as a whole should be monitored and analyzed. The monitoring information enable the system to learn from past behavior, predict future actions, and make appropriate trade offs when selecting policy actions [28]. Changes in the environment, policy adherence and violations, and changes in the system status after enforcing a policy may repeatedly trigger new policy settings. The monitoring system gathers information about events and collects performance metrics required for the governance process. The governance manager can analyze the monitored information,e.g., to find correlations between management events, system status and the expected performance. These correlations identify the actions that are most effective for specific situations.

5 Proposed governance model

In this section we introduce a business oriented governance model. This model adopts ideas of policy-driven management and optimization to build an autonomic cloud governance model. Although the approach is general enough for both SP and IP use, we here for clarity restrict the presentation of the governance model to an IP perspective.

Figure 1 shows the proposed business oriented governance model. The three main elements in the model are:

1. *BLO*
   The high level BLOs are the system inputs and they are expressed by high level objective functions and a set of constraints. An objective function is technically a utility function and it is defined in terms of system parameters and it is subjected to constraints defined by system administrators. In a sample scenario, constraints can be thresholds for performance delivered or acceptable fractions (from an IP business point of view) of SLA violations.

2. *System parameters*
   Utility functions depend on a set of well-defined system parameters. System parameters can be values provided by the monitoring system,

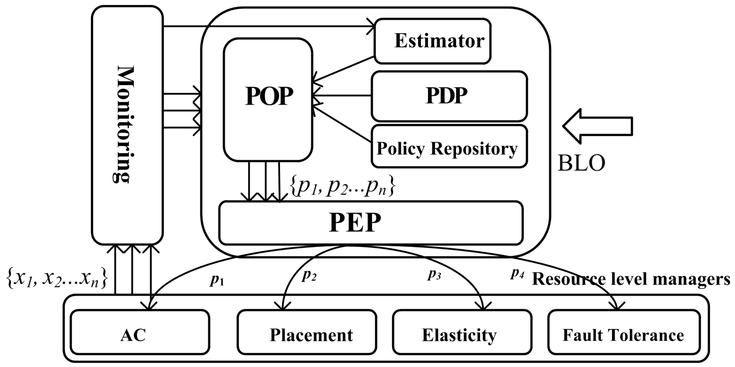


Figure 1: Global view of business oriented governance model.

used for optimization towards the BLO or for assessing how well the constraints are met. These parameters can vary from number of deployed services, the available non-allocated resources, number of violated SLA's and etc.

3. *Resource level policies* $\{p_1, \ldots, p_n\}$
   Resource level policies are used to adjust the resource level managers' behaviors. The policies are imposed as constraints or objectives on resource level managers, derived from the optimization process in the governance model. These policies carry the governance level decisions to the resource level components. Resource level managers act not only based on their local objectives but also consider new constraints enforced as governance policies. In our model policies are adjustable values used in the resource level decision making process of each manager, and they aid to meet the high level objective. The policies are adjusted by the high-level manager after monitoring the full system behavior. The idea of using adjustable policies that change over time is motivated by the dynamic behavior of individual services and the whole system performance.

The core of the governance model consists of components described below:

1. *The Policy Decision Point (PDP)* is responsible for identifying which policy decisions need to be activated in order to achieve the high level objectives as well as how these policies should be applied.

2. *The Policy Optimization Point (POP)* is responsible for setting the policy values, and real time optimization and adaptation of policy values, based on dynamic changes in service demands and system behavior.

3. *The Policy Enforcement Point (PEP)* is responsible for selecting and enforcing the right policy from the policy set to the associated resource level managers.

4. The *Policy repository* stores policies that are used in the system.

5. The *Estimator* estimates the impact of changes in the system occurring by policy setting.

The governance process is a MAPE loop [29] (Monitor, Analyze, Plan, Execute) acting as an autonomic controller. Policy sets are defined by mapping each policy in the set to a resource manager. Each policy set is evaluated by the estimator that identifies the impact of possible changes. Resource level managers which include the policies into their decision making process, e.g., as constraints or thresholds, directly adjust their actions as policy values are modified. The first step is to evaluate the system state with respect to the defined objective function. The initial policy set adopted is selected based on previous data and system history. The optimization process is executed continuously in order to improve values for the controlling parameters in the policies as these values need to reflect changes in service and system behavior.

The result is adjusted behavior of the system through an implicit coordination between low level managers. This approach automatically shifts conflict resolution between the resource managers to the governance layer. Furthermore, conflict resolution is handled during policy setting avoiding conflicts during runtime. All effects of policy enforcement: changes in system parameters, system performance, and information about managed resources, are collected by the monitoring system for further analysis. Bellow, the model is illustrated in a sample scenario. For clarity, the scenario is kept significantly simpler than a full scale real world use case.

In this scenario, an IP's BLO is to maximize its revenue without losing reputation with its customers (which is formulated as keeping the SLA violation percentage less than 5%). The utility function is given by equations 1 and 2.

$$Max\ Profit = Max(Revenue - Expense) = Max\ U()$$

where

$$U() = \sum_{i=1}^{n} R_i(t) - \bar{e}(t) - \sum_{i=1}^{n} (\hat{e}_i(t) + \check{e}_i(t))$$

$$\sum_{i=1}^{n} C_i(t) \leq p_1 \times C_{total} \qquad \forall t \in T$$

$$v \leq 5\% \quad \forall t \in T$$

where $i = \{1 \ldots n\}$ is a service. The $R_i(t)$ is the revenue from service i during the time slot t when T is the set of time slots [18]. The $\bar{e}(t)$ is the fixed cost (e.g., including investment costs). The $\hat{e}_i(t)$ and $\check{e}_i(t)$ are running cost and the SLA violation penalty cost in time t for service i. The $C_i(t)$ is the maximum capacity that can be requested for service i during time slot t. The C_{total} is the total physical capacity available and p_1 is a policy defining the provisioning rate.The value v is the percentage of SLA violations, specified in the constraints to be less than 5%.

As seen in *(eq.1)*, maximizing profit is positively correlated with the number of accepted services and negatively associated with SLA violation penalties. Hence, the applied policies should be in line with deciding whether increasing the number of accepted services or avoiding costly penalties. This relationship is independent of the fact that *(c.2)* is a constraint on the fraction of SLA violations. The main relationships between policies and managers are summarized in Table II.

Table 2: *Relationships between sample policies and low level managers*

Policy	**Behavior**	**Associated with**	**Policy Value**
p_1	Provisioning rate	Admission Control	1.25 (25% more)
p_2	Consolidation per host	Placement	4 VM/host

In the sample scenario, the behavior of admission controller is adjusted by tuning the policy p_1 for "provisioning rate". The policy p_1 is quantified by a value (1.25) instructing the AC to accept services whose aggregated maximum allowed capacity exceeds the local capacity by 25%. The policy p_2 defines the maximum VM consolidation level per host. This is the maximum number of VMs allowed to be deployed on a host in order to satisfy the BLO, in this example maximizing the revenue. The consolidation level also affects the elasticity controller (or vice versa), while changes in service demands should be supported by considering reasonable extra space on each host (or additional hosts available). SLA violations due to elasticity, result in paying costly penalties, which is in conflict with our BLO. Without loss of generality, we here assume only one type of VMs whereas a real use case would include a number of VM types with varying memory and CPU characteristics.

By enforcing policies, the AC and Placement engine operations are aligned. The result is that the entire system obtains the maximum revenue while avoids SLA violations to exceed 5%. In a full-scale scenario, policies should also be defined for other resource level managers, like elasticity controller, fault tolerance engine, or data manager. Moreover some constant values in this example, like $\check{e}_i(t)$ can be a function in a full scale use case.

6 Concluding remarks

This paper motivates the use of business oriented cloud governance to unify cloud resource management, striving towards an overall business level objective. The available solutions are reviewed with the main focus on perspectives on problem definition and formulation while introducing features and their significance. We also introduce a set of challenges for developing a business oriented governance model. A preliminary sketch of our proposed governance model is presented. Based on identified challenges, our future research will mostly focus on improving the model and making more comprehensive studies of BLO interpretation and policy mapping on low level managers.

We highlight that several challenges still need to be addressed in further studies, including:

1. Supporting a wide range of BLOs requires a broad understanding about various potential solutions to achieve each BLO. From a technical perspective, all the relationships between each possible actions of low level managers, the system parameters, and possible goals need to be defined. To define all these relationships is costly and difficult. Hence, adopting a technique that can automatically derive management policies and their relationships is of interest. Reinforcement learning and numerical or combinatorial optimization techniques are deemed suitable options, but a further analysis of these techniques in view of the governance model is still needed.

2. Exploring possible qualitative factors like trust, risk, and eco efficiency, to support more complex managerial objectives in the governance model is also under consideration. However, such factors increase the complexity of the governance process, since quantifying qualitative features like trust and risk is difficult. Thus, we first need to perform a cost/benefit analysis to select the appropriate quality factors to consider.

3. Extending and generalizing the governance model to be applied on other stakeholders like Service Providers(SPs) and brokers is also our future intention.

Acknowledgment

Financial support for this work is provided in part by the European Commission's Seventh Framework Programme ([FP7/2001-2013]) under grant agreement number 257115, OPTIMIS.

References

[1] E. Elmroth and P. Gardfjäll, "Design and evaluation of a decentralized system for Grid-wide fairshare scheduling," in *First International Conference on e-Science and Grid Computing, 2005.*, pp. 221–229, IEEE, 2006.

[2] A. Ferrer, F. Hernández, J. Tordsson, E. Elmroth, C. Zsigri, R. Sirvent, J. Guitart, R. Badia, K. Djemame, W. Ziegler, T. Dimitrakos, S. Nair, G. Kousiouris, K. Konstanteli, T. Varvarigou, B. Hudzia, A. Kipp, S. Wesner, M. Corrales, N. Forgó, T. Sharif, and C. Sheridan, "OPTIMIS: a Holistic Approach to cloud service Provisioning," 2010.

[3] M. Litoiu and M. Litoiu, "Optimizing resources in cloud, a SOA governance view," in *Proceedings of the 2010 Workshop on Governance of Technology, Information and Policies*, pp. 71–75, ACM, 2010.

[4] M. Litoiu, M. Woodside, J. Wong, J. Ng, and G. Iszlai, "A business driven cloud optimization architecture," in *Proceedings of the 2010 ACM Symposium on Applied Computing*, pp. 380–385, ACM, 2010.

[5] "Foundations of self-governing ICT infrastructures, FoSII." `http://www.infosys.tuwien.ac.at/linksites/FOSII /index.htm`.

[6] T. Pueschel, A. Anandasivam, S. Buschek, and D. Neumann, "Making Money With Clouds: Revenue Optimization Through Automated Policy Decisions," in *17th European Conference on Information Systems (ECIS 2009), Verona, Italy*, pp. 355–367, 2009.

[7] J. Perez, C. Germain-Renaud, B. Kégl, and C. Loomis, "Utility-based reinforcement learning for reactive grids," in *Autonomic Computing, 2008. ICAC'08. International Conference on*, pp. 205–206, IEEE, 2008.

[8] J. Perez, C. Germain Renaud, B. Kégl, and C. Loomis, "Multi-objective reinforcement learning for responsive grids," *Journal of Grid Computing*, vol. 8, no. 3, pp. 473–492, 2010.

[9] H. Moon, Y. Chi, and H. Hacigumus, "SLA-Aware Profit Optimization in Cloud Services via Resource Scheduling," in *2010 6th World Congress on Services*, pp. 152–153, IEEE, 2010.

[10] M. Goldszmidt, M. Budiu, Y. Zhang, and M. Pechuk, "Toward automatic policy refinement in repair services for large distributed systems," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 47–51, 2010.

[11] D. Neumann, J. Stöesser, A. Anandasivam, and N. Borissov, "SORMA - Building an Open Grid Market for Grid Resource Allocation," in *4th International Workshop, GECON 2007, Rennes, France, August 28, 2007, Proceedings, GECON*, vol. 4685 of *Lecture Notes in Computer Science*, pp. 194–200, 2007.

[12] T. Püschel, N. Borissov, M. MacÃas, D. Neumann, J. Guitart, and J. Torres, "Economically Enhanced Resource Management for Internet Service Utilities," in *Lecture Notes in Computer Science, (2007): The 8th International Conference on Web Information Systems Engineering (20Acceptance rate)*, pp. 335–348, 2007.

[13] J. Altmann, C. Courcoubetis, G. Stamoulis, M. Dramitinos, T. Rayna, M. Risch, and C. Bannink, "GridEcon: A market place for computing resources," *Grid Economics and Business Models*, pp. 185–196, 2008.

[14] J. Altmann, C. Courcoubetis, J. Darlington, and J. Cohen, "Gridecon - the economic-enhanced next-generation internet," in *4th International Workshop, GECON 2007, Rennes, France, August 28, 2007, Proceedings, GECON*, vol. 4685 of *Lecture Notes in Computer Science*, pp. 188–193, 2007.

[15] "Gridecon." `http://www.gridecon.eu`.

[16] M. Salehi and R. Buyya, "Adapting Market-Oriented Scheduling Policies for Cloud Computing," in *Algorithms and Architectures for Parallel Processing* (C.-H. Hsu, L. Yang, J. Park, and S.-S. Yeo, eds.), vol. 6081 of *Lecture Notes in Computer Science*, pp. 351–362, Springer Berlin / Heidelberg, 2010.

[17] J. Silva, L. Veiga, and P. Ferreira, "Heuristic for resources allocation on utility computing infrastructures," in *Proceedings of the 6th international workshop on Middleware for grid computing*, pp. 1–6, ACM, 2008.

[18] N. W. Paton, M. A. T. Aragão, K. Lee, A. A. A. Fernandes, and R. Sakellariou, "Optimizing utility in cloud computing through autonomic workload execution," *IEEE Data Eng. Bull.*, vol. 32, no. 1, pp. 51–58, 2009.

[19] D. M. Chess, A. Segal, and I. Whalley, "Unity: Experiences with a Prototype Autonomic Computing System," in *ICAC '04: Proceedings of the First International Conference on Autonomic Computing (ICAC'04)*, (Washington, DC, USA), pp. 140–147, IEEE Computer Society, 2004.

[20] E. Manoel, S. Brumfield, K. Converse, M. DuMont, L. Hand, G. Lilly, M. Moeller, A. Nemati, and A. Waisanen, "Provisioning On Demand: Introducing IBM Tivoli Intelligent Think Dynamic Orchestrator ," 2003.

[21] R. Das, I. Whalley, and J. Kephart, "Utility-based collaboration among autonomous agents for resource allocation in data centers," in *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pp. 1572–1579, ACM, 2006.

[22] J. Li, J. Chinneck, M. Woodside, M. Litoiu, and G. Iszlai, "Performance model driven QoS guarantees and optimization in clouds," in *Proceedings of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing*, pp. 15–22, IEEE Computer Society, 2009.

[23] J. Li, J. Chinneck, M. Woodside, and M. Litoiu, "Fast scalable optimization to configure service systems having cost and quality of service constraints," in *Proceedings of the 6th international conference on Autonomic computing*, pp. 159–168, ACM, 2009.

[24] V. Emeakaroha, I. Brandic, M. Maurer, and S. Dustdar, "Low Level Metrics to High Level SLAs-LoM2HiS framework: Bridging the gap between monitored metrics and SLA parameters in Cloud environments," in *High Performance Computing and Simulation (HPCS), 2010 International Conference on*, pp. 48–54, IEEE, 2010.

[25] M. Maurer, I. Brandic, V. Emeakaroha, and S. Dustdar, "Towards knowledge management in self-adaptable clouds," in *2010 6th World Congress on Services*, pp. 527–534, IEEE, 2010.

[26] G. Tesauro, N. Jong, R. Das, and M. Bennani, "On the use of hybrid reinforcement learning for autonomic resource allocation," *Cluster Computing*, vol. 10, no. 3, pp. 287–299, 2007.

[27] C. Li and L. Li, "A distributed decomposition policy for computational grid resource allocation optimization based on utility functions," *Microprocessors and Microsystems*, vol. 29, no. 6, pp. 261–272, 2005.

[28] R. Bahati and M. Bauer, "Adapting to run-time changes in policies driving autonomic management," in *Fourth International Conference on Autonomic and Autonomous Systems*, pp. 88–93, IEEE, 2008.

[29] IBM, "An architectural blueprint for autonomic computing," *white paper*, 2004.

Paper II

Autonomic Resource Allocation for Cloud Data Centers: A Peer to Peer Approach

M. Sedaghat, F. Hernandez-Rodriguez, and E. Elmroth

In Proceedings of the International Conference on Cloud and Autonomic Computing (IC-CAC), pp. 131-140, ACM, 2014.

Autonomic Resource Allocation for Cloud Data Centers: A Peer to Peer Approach[*]

Mina Sedaghat[†], Francisco Hernandez-Rodriguez[†], Erik Elmroth[†]

Abstract

We address the problem of resource management for large scale cloud data centers. We propose a Peer to Peer (P2P) resource management framework, comprised of a number of agents, overlayed as a scale-free network. The structural properties of the overlay, along with dividing the management responsibilities among the agents enables the management framework to be scalable in terms of both the number of physical servers and incoming Virtual Machine (VM) requests, while it is computationally feasible. While our framework is intended for use in different cloud management functionalities, e.g. admission control or fault tolerance, we focus on the problem of resource allocation in clouds. We evaluate our approach by simulating a data center with 2500 servers, striving to allocate resources to 20000 incoming VM placement requests. The simulation results indicate that by maintaining an efficient request propagation, we can achieve promising levels of performance and scalability when dealing with large number of servers and placement requests.

Keywords: Cloud computing; Peer to Peer; Resource management.

1 Introduction

The explosive growth of data centers, in terms of both size and number of servers, has greatly increased the complexity of managing data center resources. The most recent type of large scale data centers are cloud data centers, which are typically used for on-demand service provisioning. One key challenge

[*]The paper has been re-typeset to match the thesis style.

[†]Dept. of Computing Science, Umeå University, SE-901 87 Umeå, Sweden, email: {mina, francisco, elmroth}@cs.umu.se

in cloud data centers is to develop resource management frameworks and mechanisms to provide efficient resource utilization while they are also scalable and computationally feasible, with respect to the size of the data centers, the incoming load and their dynamic nature.

Most existing approaches to resource management [1–3] are highly centralized and do not scale with the number of servers in the data center. Typically, a centralized manager is required to execute the necessary complex algorithms and must also be aware of the state of all servers, which can be challenging in large and highly dynamic data centers [4].

In contrast, distributed approaches to resource management can cope with large numbers of resources without requiring centralized control. Within such approaches, the management responsibilities are divided among identical autonomic elements (nodes), helping the management structure to scale as the number of nodes increases. Global management is achieved through co-operative interactions between autonomic elements [5].

Peer to Peer (P2P) systems have proven to be scalable and robust for distributed resource management. In such systems, each peer performs a task based on locally-available information, and goal-oriented coordination among the tasks enables the system to achieve its global objective. The system thus benefits from a high degree of concurrency and decentralization of control with no central bottleneck.

However, P2P systems also face challenges due to the lack of global view of the system and not having a centralized point of reference. To compensate for this lack of global view, attempts have been made to extract and discover the required information via discovery algorithms that allow individual elements to obtain sufficient information when required.

In this paper, we address the issue of resource management in large cloud data centers, approaching it as an information discovery problem in a P2P structure. We propose a P2P resource management framework consisting of an agent community that interacts in a goal-oriented fashion. The agent community is structured as a scale-free network, enabling the agents to efficiently discover the information required for their decisions, using a simple *local search* algorithm. Our main objective is to identify a solution that is scalable both in terms of the number of servers and incoming VM requests while still being computationally feasible.

While our framework is intended to support different cloud management functionalities, e.g. admission control or fault tolerance, our primary focus is on the problem of resource allocation in clouds. As part of the work, we propose a resource allocation mechanism that aims to maximize data center utilization and profitability by ensuring high utilization of active nodes while minimizing overall power consumption by putting the remaining nodes into energy-saving mode.

We evaluated our approach by simulating a data center that has 2500 servers and must allocate resources to 20000 incoming VM placement requests. We analyzed our approach with respect to diverse performance criteria including data center utilization, profit, rejection ratio, request processing time (in terms of the number of hops per request), and scalability. We also investigated various factors that might affect performance. Our approach is shown to maintain good performance with respect to the examined criteria.

The remainder of the paper is organized as follows: Section 2 introduces the framework by describing the P2P overlay and the agent model. Section 3 presents the problem of resource allocation, the main objectives and presents a *local search* algorithm to solve the resource allocation problem. Section 4 and Section 5 discuss our experimental setup and the results obtained in the simulations conducted to evaluate the approach. Section 6 provides a brief overview of related studies, and concluding remarks are presented in Section 7.

2 Resource Management Framework

Resource management problems are often formulated as optimization problems. In order to solve them, we adopt a P2P approach, using a distributed *local search* algorithm on a population of peers, where each peer considered as a potential solution checking its neighbors in the hope of finding an improved solution.

In our design, the physical servers are structured as peers, and peers that are connected to one-another are considered neighbors. Each peer is associated with an agent that is responsible for functional tasks and local managerial decisions. Relevant information is exchanged among agents via a gossip protocol exploiting the environment formed by the peers. Each agent makes local decisions with respect to its local view and policies, and the system as a whole progresses towards the global objective via the emergent outcome of these local decisions.

2.1 Overlay Construction

In P2P systems, the overlay specifies the logical interconnections between peers. The structural properties of the overlay affect the efficiency of the discovery and propagation of information within the system, so it must be designed carefully. The problem of choosing an overlay can be formulated as a graph theoretic problem, with the physical servers (nodes) being the vertices, logical links being the edges, and nodes that are connected to one-another via an edge being neighbors that collectively form a neighborhood.

The goal is to find an overlay that is robust to failures and capable of supporting fast discovery while having a low maintenance cost (i.e. the cost of keeping nodes up-to-date about their neighbors). In graph theoretic terms, such a graph is characterized as being highly connected, sparse, with a low diameter.

Scale-free networks are a family of graphs that are widely used for structuring P2P overlays because they satisfy the criteria listed above. Scale-free networks are scalable and robust to random node failures. In addition to their robustness, these graphs have short distances between any two randomly chosen vertices and each vertex can be reached within a limited number of steps. This enables fast resource discovery, which is essential for our purposes.

In our method, the servers in the data center are structured in the form of a scale-free network that is constructed using the Barabsi Albert (BA) algorithm with a preferential attachment mechanism [6].

Such a logical structure for the resource management framework can be simply mapped into the future data center's network architectures [7], and can benefit from faster communications, resulted from the compatibility between the logical structure and the physical network structure [7].

2.2 Agent Model

On top of the P2P overlay, we build an agent community that performs functional tasks while enabling goal-oriented communication. Each agent is an autonomous entity that acts on behalf of a physical server (peer) or an application.

We associate each physical server and each application with an agent. These agents interact with each other to perform their designated tasks.

1. **Node agents** process information received from their neighbors to advance their local goal, e.g. increasing their own resource utilization or that of their neighborhood. They also direct relevant information to their neighbors. Each agent tracks information on the state of its associated physical server, including its utilization and available capacity, as well as information on its neighbors such as their state (e.g. whether they are idle, crashed or active), utilization, and free capacity.

2. **Application agents** are responsible for monitoring the application's resource demand, generating requests for more or fewer resources as the demand changes, and interacting with *nodeAgents* to deploy the new resources. The *applicationAgent* resides on one of the physical machines

on which the application is deployed, and keeps track of the VMs allocated to the application.

To solve a resource management problem, each *nodeAgent* solves a local optimization problem by searching for a locally optimal solution within its own local scope. The search proceeds iteratively from one potential solution to an improved alternative until no better solution can be found among the *nodeAgents*. Relevant information is either exchanged or distributed among agents via gossiping. The *nodeAgents* use heuristics to identify the node that offers the highest objective value.

3 Resource Allocation

We formulate the problem of placing VMs on a set of physical servers as an optimization problem.

We model the data center as a set of n physical servers, structured as a scale-free network, where each server has the capacity C_{server}. Using existing CPU/memory based capacity tools, the capacity of a server is defined in terms of number of available slots that can accommodate VMs. For clarity we assume that the servers are homogeneous, although the formulation can easily be extended for heterogeneous servers.

The data center offers k VM types, where VM-type$_{\mathrm{i}}$ (i = *1,.., k*) has capacity C_i compute units (C_i¡ C_{i+1} and $C_{\mathrm{k}} \leq C_{\mathrm{server}}$) whose price is proportional to its size.

Moreover, assume that there are m VM placement requests, where each request *j, j* = *1,..,m* demands capacity $Demand_j$. The capacity that is actually allocated to request j is denoted as Res_j. The problem is to allocate resources (slots) on the physical servers to the VMs in order to optimally fulfill a data center management objective, e.g. to maximize resource utilization and overall profit.

We define the data center utilization as the total resources allocated at time t to all VM placement requests divided by the total available capacity of the data center. The resulting optimization problem is formulated as:

$$Maximize\ U_{dc}(t) = \frac{\sum_{j=1}^{m} Res_j(t)}{n \times C_{server}} \tag{1}$$

subject to:

$$\sum_{j=1}^{m} Res_j(t) \leq n \times C_{server} \tag{2}$$

Where m is the number of placement requests, *$Res_j(t)$* is the allocated capacity for *$request_j$* at time t, n is the total number of servers (considering both idle and active servers), and C_{server} is the capacity of each server.

We can also formulate the resource allocation problem to optimize *profit*. As shown in Equation (3), we define the profit as the revenue earned from allocating the VMs minus the associated operational cost, which is formulated in terms of the cost of the servers' power consumption. The power consumption is modeled using a linear function that is shown in Equation(4), with a fixed consumption for the idle state and additional power usage proportional to the server's utilization [8]. The profit optimization problem is thus formulated as maximization of the following function:

$$Profit(t) = \sum_{j=1}^{m} Res_j(t) \times price_{Res_j} - \sum_{i=1}^{n} (P_i(t)/P_{max}) \times cost_i \tag{3}$$

This objective is also subject to constraint (2). $P_i(t)$ is the power consumption of the server at time t, which is calculated as:

$$P_i(t) = (P_{max} - P_{idle}) \times U_{node_i}(t) + P_{idle} \tag{4}$$

Here, $price_{Res_j}$ is the price of renting *$Res_j(t)$* from the data center (i.e. the data center's income), $cost_i$ is the power consumption cost for a fully utilized server, $P_i(t)$ is the power consumption of the server at time t when its processor utilization is $U_{node_i}(t)$, P_{max} is the power consumption at maximum utilization, and P_{idle} is the server's power consumption when idle.

The objective function is maximized when the aggregated power consumption of all active servers is minimized. In a homogeneous datacenter, this happens if the demand of the VMs is consolidated over the minimum servers [9].

3.1 Resource Allocation Algorithm

We propose a resource allocation algorithm, based on *local search* heuristics. This algorithm is, intrinsically, a discovery algorithm that searches for nodes according to a set of rules and specifications. A *VM placement request* traverses the network looking for a set of nodes that satisfy its resource demands. We use the term *request* as an abbreviation for *VM placement request* in the remainder of the paper.

The *nodeAgent* receiving a request selects the best potential node with sufficient capacity that can host the VM based on its locally stored information about its neighborhood. If the selected node is one of the neighbors rather than the node that received the request, the *nodeAgent* will forward the request to that neighbor so that the search for the desired resource can continue. *NodeAgents* iteratively forward the request from one potential solution to a better one until the visited *nodeAgent* is not able to find any better solution than itself or the request is expired. There are multiple heuristics that could potentially be used by the *nodeAgents* to select the best neighbor, including:

1. **Most-Utilized**: Selects the *most-utilized* node with sufficient capacity from among its neighbors (including itself). The utilization of a node is the ratio of its utilized resources to its total capacity.

2. **Least-Utilized**: Selects the *least-utilized* node with sufficient capacity from among its neighbors (including itself).

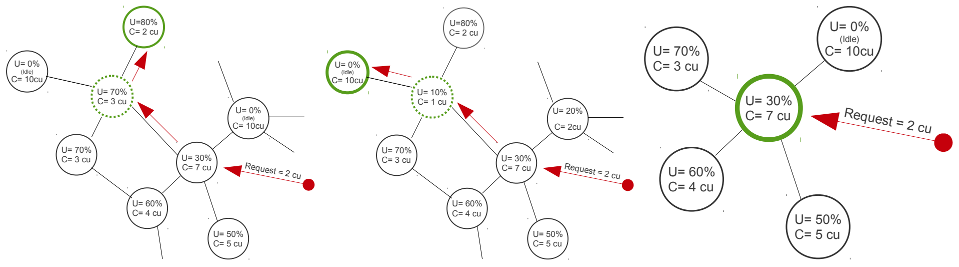
3. **First-Fit**: Selects the *first* node with sufficient capacity from among its neighbors (including itself).

Algorithm 1 Allocation

Input: Request [Demand, HTL, DecisionFlag]

Output: The location to place the request, DecisionFlag

```
On receiving a request :
if DecisionFlag = false then
  if $HTL > 0$ then
    t = identify the node w.r.t the heuristic policy and local view
    if t != $Me$ then
      if t =-1 then
        t = random neighbor from the neighborhood
      end if
      HTL = HTL - 1
      forward the request to t
    else
      location = t
      DecisionFlag = true (decision is made)
    end if
  else
    location = -1 (Location not found)
    DecisionFlag= true (Decision is made)
  end if
end if
return location and DecisionFlag
```



(a) Most-Utilized heuristic *(b)* Least-Utilized heuristic *(c)* First-Fit heuristic

Figure 1: The selection of a node for further processing of a request based on three different heuristics.

If the *nodeAgent* and one of its neighbors both offer equal objective values, the algorithm chooses the neighbor over the processing *nodeAgent*, to increase the chance of finding a better solution in the future visits and possibly skips the local optima. If none of the neighbors or the *nodeAgent* itself have sufficient capacity, the *nodeAgent* forwards the request to a random neighbor. Hence, if a request is stocked in a neighborhood with no available resources, random selection helps the request to bounce between the nodes and find a way out of the saturated neighborhood so that the search can continue.

Whenever a request is forwarded to a next *nodeAgent*, it is considered to have taken one *hop*. The total number of hops required to successfully locate a node represents the time required to process the request. We limit the request processing time using a *maximum Hops To Live (HTL)* threshold. The request is rejected if the required number of hops exceeds the *HTL*. Limiting the number of hops reduces the quality of the resulting solution and increases the number of rejections. However, it also prevents the indefinite propagation of requests.

We also assume that requests can either reach the system from a unique entry point (which is referred to as "*central entry*") or from multiple entry points ("*distributed entry*"). In both cases, they propagate from their entry points according to the same rules.

There are thus six possible combinations of heuristics and entry policies, as shown in Table 1.

3.2 Characterization of the Resource Allocation Protocol

The efficiency of the resource allocation protocol is defined with respect to our two main objectives, i.e. the maximization of data center utilization and high

Table 1: *Resource allocation policies based on different entry policies and heuristics*

Policies	**Request entry**	**Heuristics**
Central-FF	Central	First-Fit
Central-Min	Central	Min Utilization
Central-Max	Central	Max Utilization
Dist-FF	Distributed	First-Fit
Dist-Min	Distributed	Min Utilization
Dist-Max	Distributed	Max Utilization

profit. It depends on the efficiency of request propagation and the distribution of the allocated resources.

1. **Request propagation**: An efficient allocation algorithm should efficiently propagate the *VM placement requests* within the environment in a way that maximizes the likelihood of an effective visit. Effective visits are those that increase the chance of finding a node that offers a higher objective value within the *HTL* limit.

2. **Allocation distribution**: The second major factor that affects the efficiency of a resource allocation protocol is the distribution of the allocated resources. When allocated resources are sparsely distributed, it is probable that a large number of servers will have low utilization. This leads to high power consumption, increased operational costs and reduced profit. These consequences can be mitigated by distributing the allocation of resources such that a few servers are highly utilized and the rest can be put into energy saving mode.

   A sparse resource distribution also leads to fragmentation of the resources over the data center's resource pool such that the available resources on each node may be too small to place a VM even though the aggregate available capacity is still large. This can cause increased request rejection and decreased utilization.

3.3 Optimal Re-consolidation

As discussed in Section 3.2, the distribution of the allocated resources directly affects the performance of the allocation algorithm. The ultimate allocation is an emergent consequence of the heuristic adopted by the *nodeAgents*. In addition to the *nodeAgent's* heuristic decisions, the frequent arrivals and terminations of VMs can also lead to a highly sub-optimal distribution of allocated resources over time. Therefore, it can sometimes be advantageous to migrate a previously allocated VM from one node to another, either to switch off a server or to optimize the allocation distribution to open up space for larger VMs.

Algorithm 2 Re-consolidation

```
if $MyLoad$ ¡ Re-consolidationThreshold then
  for i:=1 to NumberVMsDeployedOnMe do
    request = initiate a request for $VM_i$ [Demand$_i$, $HTL$, false]
    newHost = Allocation (request)
    if newHost != - 1 then
      Migrate $VM_i$ to newHost
    end if
  end for
end if
Recalculate $MyLoad$
if $MyLoad = 0$ then
  Set $Me$ into energy saving mode
end if
```

This process is also known as *re-consolidation* of currently deployed VMs with the goal of increasing a node's utilization and potentially reducing the incidence of rejections due to resource fragmentation. To perform *re-consolidation*, *nodeAgents* representing lightly loaded servers autonomously or regularly migrate their loads to more heavily loaded nodes by initiating a request, similar to the initial placement request, for each of their deployed VMs. This request searches for the most highly utilized node with sufficient capacity to be the new host for the VM. If all the node's VMs are migrated successfully to other nodes, the node can be switched into a power saving mode. Re-consolidation also makes it easier to accommodate larger VMs and reduces the likelihood of rejection.

During re-consolidation often a performance impact can be expected. This impact can be modeled [10] and be taken into consideration before deciding on the re-consolidation. However, it has been shown that advancements in virtualization techniques [11] and technologies effectively reduce the performance overheads and its associated impacts [12]. Modeling this impact is not the main focus of this study, however we can simply extend our model to perform a cost-benefit analysis before re-consolidation, considering the overhead costs.

4 Experimental Setup

This section describes an evaluation of the performance of the proposed approach through a simulation of a data center with 2500 physical nodes. We simulated our framework in the Netlogo environment and built our P2P overlay using the scale-free network model as implemented by [13].

Each physical node was associated with a *nodeAgent* and assumed to be capable of serving VMs of different types. The maximum capacity of each physical node was set to 10 compute units and the provider was assumed to offer 10 VM types with capacities ranging from 1 to 10 compute units. The price of a VM providing 1 compute unit was 0.01$ per time unit, similar to that for a Linux micro reserved-instance in Amazon's EC2 system. The prices of larger VMs were proportional to their capacity.

We assumed a total of around 20000 incoming VM placement requests, arriving the system following a Poisson arrival rate. Each request had a capacity demand (in compute units) that was selected at random from the set {1,...,10} and was mapped to a VM type. The VMs were deployed and terminated over the course of each simulation. Services running in clouds usually have an indefinite lifetime, so we modeled VM lifetime using a normally distributed random variable in order to eliminate the potential for systematic bias associated with specific application types and to represent the diversity of applications that may be deployed in a cloud data center.

In order to avoid infinite request forwarding in the environment, we constrained the number of hops per request to $HTL = 20$ *hops*.

The *re-consolidation threshold*, i.e. the load at which a node's resources are considered for re-consolidation, was set to 40% of the node's total capacity. Simulations were allowed to run for 20000 time units with each time unit representing 0.1 sec of simulation time and one hour of resource usage.

4.1 Performance Parameters

We evaluated our approach with respect to the following performance parameters:

1. **Data center utilization ($U_{dc}(t)$):** This variable represents the utilization of the data center. It is defined as the total capacity used by the allocated VMs at time t relative to the total available capacity in the data center. Data center utilization is calculated using Equation (1), which was introduced in Section 3.

2. **Average node utilization ($U_{node}(t)$):**

$$U_{nodes}(t) = \frac{\sum_{j=1}^{m} Res_j(t)}{n_{active} \times C_{server}} \tag{5}$$

   where n_{active} is the number of active nodes in the environment. This metric provides insight into the distribution of allocations and how efficiently the currently active servers are being utilized. It is directly proportional to the system's power consumption and the associated costs.

3. **Number of hops:** This is the number of steps required to locate a node that has the capacity required by the VM request. This metric measures how quickly the algorithm can locate a suitable node and respond to a request, and can be compared to the computation time in centralized approaches.

4. **Rejection ratio:** This is the proportion of request demands that are not satisfied.

$$RR(t) = \frac{\sum_{j=1}^{m} Demand_j(t) - \sum_{j=1}^{m} Res_j(t)}{\sum_{j=1}^{m} Demand_j(t)} \tag{6}$$

   Rejections may occur for various reasons, including:

   - A failure to locate a suitable resource within the *HTL* limit.
   - A lack of sufficient resources to serve the request.
   - Fragmentation of the resource pool.

5. **Profit:** This represents the revenue of the data center with respect to the service provided and its operational costs. The profit is calculated using Equation (3), introduced in Section 3.

5 Results and Discussion

This section describes how the performance is influenced by two key properties, namely request propagation and allocation distribution, and their impact on allocation policies. We evaluate performance in terms of the five metrics introduced above, *Data center utilization*, *Node utilization*, *Number of hops*, *Rejection ratio* and *Profit*. Finally, we study the scalability of the approach when each of the 6 allocation policies is adopted.

5.1 Impact of Request Propagation on Performance

In the first series of experiments, we studied the impact of request propagation on the performance of the resource allocation mechanism. Request propagation can be affected by the entry of requests to the system, the constraints imposed by *HTL* threshold, and modifications of the overlay topology. Due to paper limits, we only discuss the impact of requests' entry on the performance.

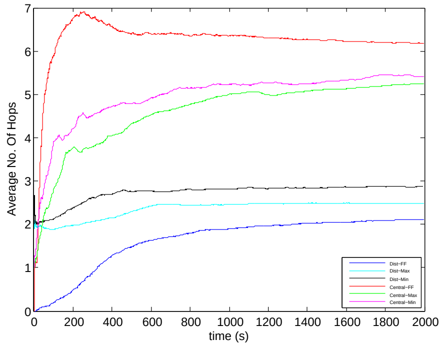
To determine how entry policy affects system performance, we compared the impact of adopting *central* and *distributed* entry policies (see Section 3.1) for VM requests on system performance.

Figure 2 shows the number of hops, rejection ratio, data center utilization, node utilization, and profit for each of the policies listed in Table 1. In general, *distributed entry* provides better request propagation, requiring fewer hops to place the VM. This is because *distributed entry* of requests automatically increases the probability of request propagation to a 'better' node and thus reduces the number of hops. The lower the number of hops, the lower the likelihood of exceeding the *HTL* and thus the lower the likelihood of request rejection. Reducing rejection ratios also increases data center utilization and profits. However, *distributed entry* generally produces lower node utilization ($U_{node}(t)$) than *centralized entry* because it results in a more sparse placement of VMs.

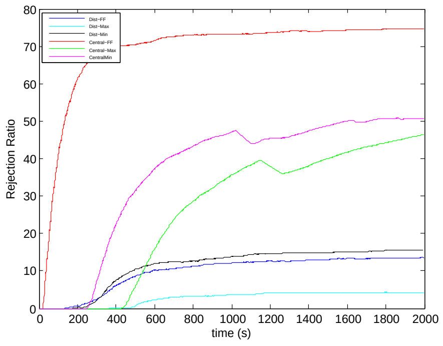
Central entry requires more hops to place a VM, leading to a higher rejection ratio and reduced data center utilization. That is to say, such policies suffer from *weak request propagation* in comparison to *distributed entry* alternatives. When requests are bound by a *central entry* policy, a small proportion of nodes experience a large number of visits (especially those in the vicinity of the entry node) while others are never visited. Consequently, the VMs tend to cluster in the neighborhood of the entry node. After a while, these nodes become fully loaded because they are so frequently visited, and become unable to accept new VMs. Subsequent requests must therefore travel beyond the saturated neighborhood, reducing the likelihood of successful allocation within the *HTL*. This in turn increases the rejection ratio. Because some fraction of the nodes can never be reached within the *HTL*, data center utilization is reduced and profit decreases.

5.2 Impact of Allocation Distribution on Performance

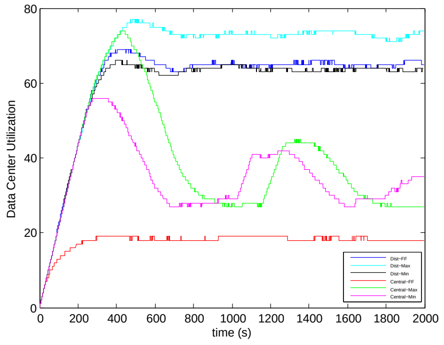
The distribution of allocated resources over the data center resource pool is the second major factor that affects the performance. It is determined by the local heuristic decisions of each *nodeAgent* in conjunction with the system's request entry policy. To determine the impact of allocation distribution on



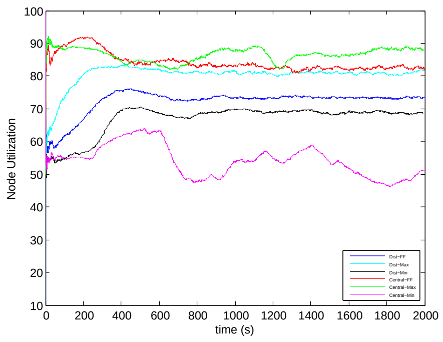
(a) Number of hops



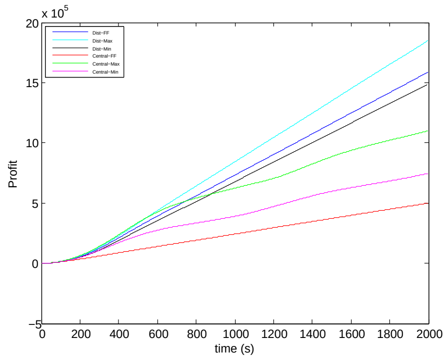
(b) Rejection ratio



(c) Data center utilization



(d) Node utilization



(e) Profit

Figure 2: The effects of different allocation policies in a data center with 2500 nodes.

performance, we compared the 6 policies presented in Table 1 when used in conjunction with 3 heuristics: *Least Utilization*, *Most Utilization* and *First-Fit*.

The average node utilization, $U_{node}(t)$, is a useful metric for analyzing the distribution of allocated resources. For a given amount of allocated capacity, we can either distribute the allocations sparsely to provide a large number of lightly-loaded nodes, or we can consolidate them across a small number of nodes with high $U_{node}(t)$ values.

Figures 2a and 2b show the number of hops and the rejection ratios for each allocation policy. Of the six policies *Central-FF* requires the highest average number of hops to find a resource and place the VM, and it also has the highest rejection ratio. This can be explained by the fact that the weak request propagation of the *central entry* policy along with the saturation of the entry node's neighborhood caused by the *First-Fit* heuristic forces requests to make extra ineffective hops. This increases the required number of hops needed to locate a resource, causing requests to exceed the *HTL* and be rejected. The consequence of this is clearly shown in Figures 2b and 2c: the rejection ratio increases and data center utilization decreases. However, because allocated VMs are densely distributed over a small number of nodes within the vicinity of the entry node, this policy achieves high utilization values $U_{node}(t)$ for the active nodes, although the number of active nodes is limited. It is important to point out that data center utilization and node utilization $U_{node}(t)$ are not necessarily correlated. For example, as shown in Figure 2, while the overall data center utilization rate when using the *Central-FF* algorithm is only 20%, the 20% of active nodes have an average utilization level of 80% (i.e. $U_{dc}(t_1) = 20\%, U_{node}(t_1) = 80\%$). Due to the high rejection ratio, this policy produces the lowest data center utilization and thus generates the lowest profit of all the policies evaluated.

On the other hand, we can see that the *Dist-FF* policy requires the fewest hops because it generates no saturation and the allocated VMs are distributed across the entire data center. The number of hops in this case is low because the *First-fit* heuristic merely selects the first node with available capacity and does not search further. The low number of hops results in a low rejection ratio and a fairly high data center utilization and profit.

Central-Min has the second highest number of hops and rejections. This is because the *Least-utilized* heuristic generates a lot of lightly loaded nodes due to its policy of placing VMs on loads with low utilization. In this situation, the resources are fragmented over the data center's resource pool and it becomes harder to place large requests. Consequently, the number of rejections increases because each available fraction is too small to place a large request, even though the total available capacity remains high. The *central-entry* of the requests is also another reason for the high rejection ratio under the *Central-Min* policy due to its weak request propagation. As shown in Figure 2d, this policy has a low node utilization $U_{node}(t)$ value because it frequently starts idle nodes and generates a large number of lightly utilized active nodes. The low data center utilization and high number of active nodes makes this policy one of the least profitable options.

Dist-Min policy provides better request propagation due to the *distributed entry* of requests, and thus increases the data center utilization relative to its *Central-Min* counterpart. However, the *Least-utilized* heuristic, which is common to both the *Dist-Min* and the *Central-Min* policies, causes resource fragmentation and increases the number of rejections.

As shown in Figure 2, *Dist-Max* achieves the best performance of the tested policies. The *Most-utilized* function selectively places VMs on nodes with sufficient capacity and the highest overall utilization. This heuristic automatically avoids the fragmentation of the resources by consolidating as many VMs as possible onto each active node, and can therefore accommodate more demand than the *Least-utilized* approach. It also avoids activating idle nodes because it tries to place the VMs onto currently active nodes to increase their utilization. This is why *Dist-Max* is the most profitable policy: it achieves the greatest possible data center utilization with the lowest possible number of active physical servers.

In summary, policies based on the *distributed entry* of requests offer better request propagation, yielding better performance and higher profits. The heuristic that offers the best performance and profitability is *Most-utilized*, followed by *First-fit*. Policies based on the *Least-utilized* heuristic has the lowest performance when the main objective is profit and having high consolidation. However, policies adopting the *Least-utilized* heuristics can be effective when other objectives such as load balancing is the main concern.

5.3 The Impact of Re-consolidation on Performance

In the previous section we showed that *Least-utilized* heuristics generate lightly loaded servers, causing fragmentation of the resource pool that leads to low data center utilization and reduces profits. In addition, the frequent arrival and termination of VMs can also lead to a far from optimal allocation distribution that may affect the performance of the resource allocation mechanism over time. In this section, we study how the re-consolidation of VMs can improve performance in such situations.

Table 2 shows the total number of servers put into power saving mode after re-consolidation during the simulation time when the allocation algorithm used in the simulation follows each of the six above-mentioned policies. The number of servers that undergo re-consolidation under *central entry* policies is low relative to that for *distributed entry* policies. This is because in *central entry* policies, most of the allocations are densely populated within the entry node's vicinity and resources are not sparsely allocated. Consequently, the number of nodes that are lightly loaded enough to trigger the re-consolidation process is much lower than under *distributed entry* policies.

Table 2: *Number of nodes put into hibernation after re-consolidation*

Policies	**# nodes hibernated due to re-consolidation**
Central-FF	35
Central-Max	202
Central-Min	2693
Dist-Max	1030
Dist-FF	2147
Dist-Min	3679

Both Table 2 and Figure 3 show that *Least-utilized* policies benefit the most from the re-consolidation process. It is also clear that the resource distributions generated by *Most-utilized* policies are those that change the least following re-consolidation. This is because these policies preferentially deploy VMs onto highly loaded nodes in the first place.

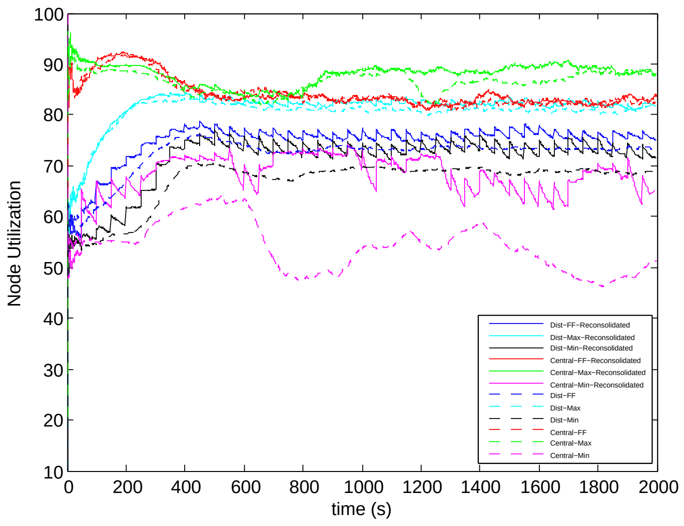


Figure 3: Impact of re-consolidation on node utilization.

The most significant impact of re-consolidation is on the node utilization, $U_{node}(t)$, because it packs the allocations onto a smaller number of servers and thus reduces the number of active servers while increasing their utilization. We should however note that not all re-consolidations lead to servers being powered down. The *nodeAgent* may look for potential hosts to migrate its deployed VMs one by one, but this does not necessarily mean that appropriate new locations will be found for all of them.

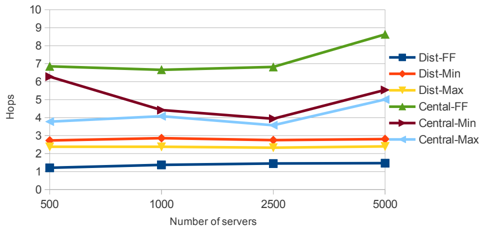
5.4 Scalability Analysis

To study the scalability of our approach, we performed simulations for data centers with 500, 1000, 2500 and 5000 servers, with 800, 1750, 4500, and 9000 VM requests, respectively. In this experiment, VMs were not re-consolidated and also not terminated so that we could study the scalability in an extreme case where all of the resources in the data center are saturated.

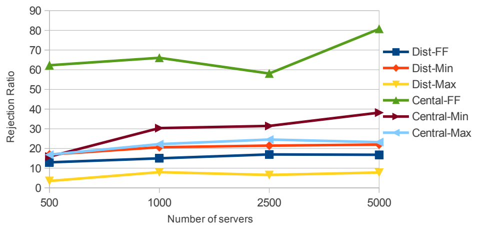
Figure 4 shows the performance of each policy with respect to server count. It is clear that the performance does not depend on system size provided that the allocation policies maintain adequate request propagation. For allocation policies based on *distributed entry*, performance metrics such as the *number of hops*, *rejection ratio* and *data center utilization* remain constant as the number of servers increases. This is because in systems with large numbers of servers, the main concern is to ensure that all nodes can be reached efficiently within an acceptable number of hops. This is straightforward when using *distributed entry* policies due to their favorable request propagation properties. Because requests are propagated efficiently, increasing the number of servers does not increase the number of hops or the frequency of VM rejection. As discussed above, data center utilization is highly dependent on the rejection ratio; because the rejection ratio is independent of the server count in this case, the data center utilization is as well.

However, this approach is not scalable when it is applied in conjunction with an allocation policy that has weak request propagation such as *Central-FF*. This is because such policies prevent requests from reaching most of the servers in the system. Therefore, as the number of servers increases, more requests are rejected and more servers remain un-utilized.

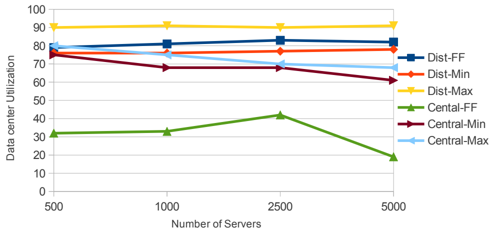
The *Node utilization* metric does not capture the dynamics of the system when the size is increased because it just expresses the utilization of active nodes.



(a) Number of hops



(b) Rejection ratio



(c) Data center utilization

Figure 4: Scalability of the approach with respect to increasing numbers of servers

6 Related Work

There are a number of researches that are related to our study.

The first group of studies focused on the problem of VM placement adopting a centralized approaches, tackling the problem by formulating it as a knapsack or constraint satisfaction problem and generating solutions using integer programming methods [2, 14, 15]. These methods provide high quality solutions for limited numbers of servers and applications but must compromise on solution quality when applied to large scale data centers in order to achieve computational tractability.

The second group of related research includes studies that examined P2P approaches for resource management in cloud environments. Barabagallo et al. [16] modeled a data center as a P2P network of self-organizing nodes that collaborate with one-another using bio-inspired algorithms. This collaboration allows the nodes to redistribute the load among servers in order to increase the system's energy efficiency. Their idea is to have a number of entities known as scouts that investigate and gather information about virtual machines in the data center. This information is then used by other virtual machines to initiate migrations in order to redistribute the overall load. Their approach is related to our work on optimal re-consolidation. However, we have shown that a P2P approach can be adopted for wider problems such as resource allocation, and that re-consolidation is just one component of the broader resource allocation problem. Our work also deals with business goals such as utilization and profit, and yields improvements in energy efficiency as a consequence of achieving these goals.

Wuhib et al. [17] used a gossip protocol for dynamic resource management in clouds. In their protocol, the nodes interact with a subset of other nodes via small messages. These messages allow nodes to exchange state information and then compute a new configuration with the goal of maximizing cloud utility. If the gain from a new configuration outweighs the cost of change, they adopt the change and update their local state. This approach differs from ours in terms of the type, the purpose of the interactions and gossip: the authors' main focus is on the fairness of their allocations whereas we focused primarily on data center utilization and profit.

Marzolla et al. [18] also adopted gossiping for server consolidation in order to decrease power consumption. Their main focus is on the migration of arbitrary placed applications as a way of decreasing power consumption. As mentioned above, we approach the migration (re-consolidation) process as part of a larger solution to optimal resource allocation.

7 Conclusions

In this paper we discussed a novel approach to perform VM placements in cloud data centers. Our approach benefits from high degree of concurrency and decentralization of control with no central bottleneck. Our main contributions are:

1. A new formulation of resource management problem through a P2P framework.

2. Proposing a P2P overlay based on scale-free network for robust and efficient discovery of the most suitable potential server for VM placement.

3. A resource allocation algorithm based on local search, designed to maximize data center utilization and profitability. The algorithm ensures high utilization of active nodes while minimizing overall power consumption by putting the remaining nodes into energy saving mode.

We investigated the impact of different heuristics on the quality of the resulting allocations, with respect to the specified objectives of maximizing data center utilization and profitability. We also studied the scalability of our approach by evaluating its performance with different numbers of servers. Our approach was shown to be scalable up to at least systems of 5000 nodes with 9000 incoming placement requests arriving during the simulation time when using policies that allow for efficient request propagation.

We also present a re-consolidation process as a component of the broader resource allocation process. This enables the optimal re-allocation of currently running VMs. The re-consolidation process is designed to redistribute allocations among the servers in a data center in order to utilize the active nodes more efficiently in cases where the existing allocation has become sub-optimal. Efficient utilization makes it possible to switch off lightly loaded servers and reduce the center's overall power consumption, thereby increasing profits.

Acknowledgment

Financial support for this work is provided in part by the Swedish Research Council (VR) under contract number C0590801 for the project Cloud Control and the Swedish Government's strategic research project eSSENCE.

References

[1] A. J. Ferrer, F. Hernández, J. Tordsson, E. Elmroth, A. Ali-Eldin, C. Zsigri, R. Sirvent, J. Guitart, R. M. Badia, K. Djemame, *et al.*, "Optimis: A holistic approach to cloud service provisioning," *Future Generation Computer Systems*, vol. 28, no. 1, pp. 66–77, 2012.

[2] W. Li, J. Tordsson, and E. Elmroth, "Virtual machine placement for predictable and time-constrained peak loads," in *Economics of Grids, Clouds, Systems, and Services*, pp. 120–134, Springer, 2012.

[3] T. Püschel, N. Borissov, M. Macías, D. Neumann, J. Guitart, and J. Torres, "Economically enhanced resource management for internet service utilities," in *Web Information Systems Engineering–WISE 2007*, pp. 335–348, Springer, 2007.

[4] C. Mastroianni, M. Meo, and G. Papuzzo, "Self-economy in cloud data centers: statistical assignment and migration of virtual machines," in *Euro-Par 2011 Parallel Processing*, pp. 407–418, Springer, 2011.

[5] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[6] A.-L. Barabási and R. Albert, "Emergence of scaling in random networks," *science*, vol. 286, no. 5439, pp. 509–512, 1999.

[7] J.-Y. Shin, B. Wong, and E. G. Sirer, "Small-world datacenters," in *Proceedings of the 2nd ACM Symposium on Cloud Computing*, p. 2, ACM, 2011.

[8] L. Minas and B. Ellison, "The problem of power consumption in servers," *Intel Corporation. Dr. Dobb's*, 2009.

[9] F. Wuhib, R. Yanggratoke, and R. Stadler, "Allocating compute and network resources under management objectives in large-scale clouds," *Journal of Network and Systems Management*, pp. 1–26, 2013.

[10] A. Verma, G. Kumar, R. Koller, and A. Sen, "Cosmig: Modeling the impact of reconfiguration in a cloud," in *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2011 IEEE 19th International Symposium on*, pp. 3–11, IEEE, 2011.

[11] P. Svärd, J. Tordsson, E. Elmroth, S. Walsh, and B. Hudzia, "The noble art of live vm migration -principles and performance of precopy and postcopy migration of demanding workloads,"

[12] A. Beloglazov, J. Abawajy, and R. Buyya, "Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing," *Future Generation Computer Systems*, vol. 28, no. 5, pp. 755–768, 2012.

[13] U. Wilensky, "Netlogo preferential attachment model," *Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, Illinois, http://ccl. northwestern. edu/netlogo/models/PreferentialAttachment*, 2005.

[14] D. Breitgand, A. Marashini, and J. Tordsson, "Policy-driven service placement optimization in federated clouds," *IBM Research Division, Tech. Rep*, 2011.

[15] S. Chaisiri, B.-S. Lee, and D. Niyato, "Optimal virtual machine placement across multiple cloud providers," in *Services Computing Conference, 2009. APSCC 2009. IEEE Asia-Pacific*, pp. 103–110, IEEE, 2009.

[16] D. Barbagallo, E. Di Nitto, D. J. Dubois, and R. Mirandola, "A bio-inspired algorithm for energy optimization in a self-organizing data center," in *Self-Organizing Architectures*, pp. 127–151, Springer, 2010.

[17] F. Wuhib, R. Stadler, and M. Spreitzer, “A gossip protocol for dynamic resource management in large cloud environments,” *IEEE Transactions on Network and Service Management*, vol. 9, no. 2, pp. 213–225, 2012.

[18] M. Marzolla, O. Babaoglu, and F. Panzieri, “Server consolidation in clouds through gossiping,” in *2011 IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM),*, pp. 1–6, IEEE, 2011.

Paper III

Extended version of the paper:

Divide the Task, Multiply the Outcome: Cooperative VM Consolidation

M. Sedaghat, F. Hernandez-Rodriguez, E. Elmroth, and S. Girdzijauskas

In Proceedings of the 6^{th} *IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 300-305, IEEE Computer Society, 2014.

Divide the Task, Multiply the Outcome: Cooperative VM Consolidation[*]

Mina Sedaghat[†], Francisco Hernandez-Rodriguez[†], Erik Elmroth[†], Sarunas Girdzijauskas[‡]

Abstract

Efficient resource utilization is one of the main concerns of cloud providers, as it has a direct impact on energy costs and thus their revenue. Virtual machine (VM) consolidation is one the common techniques, used by infrastructure providers to efficiently utilize their resources. However, when it comes to large-scale infrastructures, consolidation decisions become computationally complex, since VMs are multi-dimensional entities with changing demand and unknown lifetime, and users often overestimate their actual demand. These uncertainties urges the system to take consolidation decisions continuously in a real time manner.

In this work, we investigate a decentralized approach for VM consolidation using Peer to Peer (P2P) principles. We investigate the opportunities offered by P2P systems, as scalable and robust management structures, to address VM consolidation concerns. We present a P2P consolidation protocol, considering the dimensionality of resources and dynamicity of the environment. The protocol benefits from concurrency and decentralization of control and it uses a dimension aware decision function for efficient consolidation. We evaluate the protocol through simulation of 100,000 physical machines and 200,000 VM requests. Results demonstrate the potentials and advantages of using a P2P structure to make resource management decisions in large scale data centers. They show that the P2P approach is feasible and scalable and produces resource utilization of 75% when the consolidation aim is 90%.

[*]The paper has been re-typeset to match the thesis style.

[†]Dept. of Computing Science, Umeå University, SE-901 87 Umeå, Sweden, email: {mina, francisco, elmroth}@cs.umu.se

[‡]KTH - Royal Institute of Technology, Sweden, email: sarunasg@kth.se

Keywords: Cloud computing; Peer to Peer; Gossip protocols; VM consolidation; Resource management.

1 Introduction

Cloud providers are widely using the virtualization technologies for the efficient utilization of their available resources. Virtual Machines (VMs) are treated as blocks that can be put together on a Physical Machine (PM) or can be moved from one PM to the other to maintain a high resource utilization in a data center. Consolidation of multiple VMs on a single machine helps cloud providers to increase their resource utilization and decrease their power consumption, specially as users often overestimate their actual demand. It also helps opening up capacity to run services with special constraints or the ones that are larger to fit in a small fragmented resource.

However, the fact that VMs are serving a changing demand and they have unknown arrivals and lifetimes forces the system to continuously make the consolidation decisions and re-assign the resources in a real time manner. It should be noted that both VMs and PMs are multi-dimensional entities, i.e. they are defined in terms of CPU and memory capacity, and they have specific shapes. Thus, maintaing the efficiency of utilization when the entities are multi-dimensional and their shapes are changing over-time can be challenging.

Moreover, the applications and management decisions become more complex and larger in scale, such that the traditional centralized or hierarchical approaches cannot scale with the number of PMs in the data center. To support scalability, they either need to compromise the quality of the solution to the resource management problem, to keep the response time within an acceptable time frame or alternatively partition the infrastructure statically. The latter restricts the system from using the resources efficiently due to the lack of coordination between partitions. Centralized approaches also require a continuous fine grained monitoring data which is often huge and expensive to collect and process [1].

However, decentralized approaches allow the complex resource management decisions to be taken in collaboration, by a number of autonomous entities working toward a management objective. In such systems, the objective is achieved as the emergent behavior of a number of autonomous entities, acting as processing units, making management decisions within their own local scope. Besides, such systems are more robust since each autonomous entity operates independently from others' failures or departures.

In this paper, we investigate how resource management problem in cloud data centers can be formulated and addressed using a decentralized approach.

We specifically tackle the VM consolidation problem, to achieve energy efficiency and increase resource utilization. Extending our resource management framework introduced in [2], we propose a P2P gossip-based protocol for VM placement and consolidation, considering the multi-dimensionality of the PMs and VMs. Each peer in the P2P framework, cooperates with its fellow peers, to improve its status with a new state of a greater value, thus moving toward a more efficient state. The peer uses a dimension aware decision function to quantify its state and determine the right actions. Besides, each peer only monitors its own resources and a small-sized metadata is exchanged only to the peer's immediate neighbors. We also perform an extensive study on feasibility and performance of the proposed approach when scale matters.

We show that the P2P approach is feasible and scalable and it produces an almost-optimal VM placement for the experimented scale of 100,000 PMs and 200,000 VM requests when the load is dynamic. The system benefits from a high degree of concurrency and decentralization of control with no central bottleneck. This help the system to have a low computation time in making placement decisions for the mentioned scale. This advantage is essential for real time management of a dynamic environment such as cloud infrastructures. We also believe that delegating the complex decisions to the autonomous entities would eliminate cumbersome configuration settings that are required in the centralized approaches. It is an important advantage, since these configuration values, i.e, low utilization thresholds, offload trigger points or monitoring intervals are usually hard to devise, not applicable for all the data center's entities and they are sensitive to the load changes.

2 Consolidation problem

Assume that a data center consists of n PMs. Each PM has a CPU and memory capacity of C_{PM} and M_{PM}. The CPU and memory utilization of each PM i at time t is denoted as $c_i(t)$ and $c_i(t)$, respectively. It is assumed that all PMs are homogeneous, although the formulation can also be generalized for heterogeneous machines. The data center offers l VM types, where each VM has an expected computational and memory capacity of CPU_l and Mem_l. These VMs are categorized in 3 different types of *compute optimized*, *memory optimized* and *general purpose*, due to their usability scenario. It is also assumed that m VM placement request arrive to the system during the data center operation time. Each request j demands a specific VM type l and consumes a $CPUDemand_j(t)$, $MemDemand_j(t)$ at each timestep.

The goal is to achieve energy efficiency, $E(t)$, in the data center by minimizing the total power consumption, modeled as a function shown in Equation (1).

$$Minimize\ E(t) = \sum_{i=1}^{n_{active}} P_i(t) \tag{1}$$

subject to:

$$\sum_{j=1}^{m} Res_j(t) \leq n_{active} \times (\alpha \times Capacity_{PM}) \tag{2}$$

where $Res_j(t)$ is the utilized capacity by *VM request*$_j$ at time t, n_{active} is the number of non-idle PMs, α is a utilization factor, defined to avoid performance degradations caused by interferences among the consolidated VMs, and finally, $Capacity_{PM}$ is the capacity of the PM in terms of CPU and memory. For the sake of brevity, we generalize the notations of $Res_j(t)$ and $Capacity_{PM}$ to address the concept of *resource*. However, whenever these values are referred in the paper, we calculate the value for each resource, CPU or memory, independently with respect to their relevant values. Moreover, P_i(t) is the consumed power of PM_i, and it is a linear function of the fixed consumption for PM when it is in idle state and additional power usage proportional to PM's CPU utilization [3] and it is calculated by Equation (3).

$$P_i(t) = (P_{max} - P_{idle}) \times c_i(t) + P_{idle} \tag{3}$$

where P_{max} is the power consumption at maximum performance and P_{idle} is the PM's power consumption when idle.

One important point to be considered is that resources are multi-dimensional. Hence, for a mixed workload, consisting of both compute and memory optimized VMs, the placement algorithm should be dimension aware, meaning that the proportionality of resource usage (CPU vs. memory) in each PM should be considered while deciding on the placement [4,5]. To best utilize the capacity of a physical machine along two dimensions (CPU/Memory), it is desirable that at each point of time, the resources being proportionally utilized in both dimensions. The unbalanced utilization of the resources in each physical machine will lead to wasting the space in one of the dimensions. Therefore, to ensure the efficient utilization, we also strive to minimize an imbalance factor, formulated as:

$$Minimize\ Imb_{dc}(t) = (\sum_{i=1}^{n} |c_i(t) - m_i(t)|)/n \tag{4}$$

where $Imb_{dc}(t)$ is the average of data center's imbalance at time t, n is the total number of PMs, both idle and non-idle, $c_i(t)$ and $m_i(t)$ are the relative values for CPU and memory utilization of PM_i at time t.

3 VM placement through gossiping

We consider the VM placement problem as a distributed decision making process. We introduce a decentralized gossip-based protocol, where decisions are continuously being made between random pairs of cooperative agents, trying to improve a common value. The common value is defined as the total imbalance, *Imb(t)*, of each pair at the time of decision making and the goal is to reduce this imbalance by redistributing the VMs among them. The cooperative approach among two agents prevents the undesirable bounces of VMs and ensures a stable state, which is essential to avoid the redundant migrations.

3.1 General Architecture

A data center is a collection of PMs. In our design, each PM is considered as a peer in a P2P network. Peers are logically connected by an overlay network. The overlay is built and maintained by a peer sampling service, known as newscast [6]. Peer sampling service periodically provides each peer with a list of peers to be considered as neighbors. Each peer at each timestep only knows about k random neighbors, shaping its local view.

Each peer locally runs a resource agent responsible for monitoring its associated PM's state (CPU and memory consumption), communicating with the neighbors and processing the information received from the neighbors. Peers communicate in a gossip-based fashion via small messages that represent their state. The details about the general architecture is discussed in [2].

3.2 Design concerns

To investigate the most suitable solution, we review the common questions and enumerate some of the issues that should be addressed during the design.

1. **When should a consolidation be performed?**

   In common practices, the re-consolidation process is triggered by an event, i.e. when a PM detects a violation of an under or over-utilization threshold. However, event based algorithms, require configuring thresholds which is often tricky and complex and need a good understanding about the system and the changes in load. In such algorithms, setting the threshold too low, may lead to losing the chance of efficient consolidation of VMs, while setting it too high, may end up moving VMs constantly from one machine to the other.

   Thresholds may also be susceptible to changes since the system dynamics are changing over time. In addition to that, they are usually defined as absolute values, so a small deviation from the threshold would disqualify the PM to be triggered for re-consolidation. Hence, there would be cases, that the load of two machines can be accommodated in one, but since the load in none of the PMs fall below the threshold, this consolidation would never happen.

The autonomy of the peers in a P2P structure allows the system to avoid the complexity of threshold setting. Using P2P structure, the peers involved in the decision process use their real time state to decide if they can benefit from a re-distribution of their VMs or not. If both peers realize that their load can be accommodated in one, within a reasonable cost of migration, then potentially a re-consolidation process can be triggered when the P2P interactions are converged. This way re-consolidation is not bound to the thresholds but it would be planned dynamically based on the peers' states. Hence, adopting P2P approach decreases the need for threshold setting and also increases the chance of triggering more efficient re-consolidations.

2. **How to re-consolidate?**

   The second question is how the re-consolidation should be performed? One strategy can be to incrementally migrate the VMs from a low-utilized machine to a PM with available capacity, with the hope of future release. The other option is to migrate VMs only when the complete release of the PM is guaranteed. Each of these two approaches have their own advantages and weaknesses that we will discuss in next sections.

3. **Consolidation consequences:**

   Different works have studied the impacts of consolidating multiple VMs on the same machine on the performance [4, 7]. The impacts are usually due to the interferences and contentions among VMs, such as cache contentions, conflicts at functional units of CPU, disk write buffer or disk scheduling conflicts [7]. However, consolidation of multiple VMs also increases the probability of overloading the PMs when the load of the VMs are changing overtime.

   The PM overloads can be handled in different ways, such as service differentiation [8, 9] or application brownouts [10], but they are usually handled through PM offloads via migration. Offload processes are often costly. The cost is due to the performance impacts or possible SLA violations and also the additional management process required to decide which VMs to be migrated to where. Based on the above-mentioned impacts, we can argue that the best strategy for VM placement and re-consolidation is the one that accommodates the demand on the fewest PMs while minimizing the over-consolidation consequences. These consequences can be listed as the longer overload time experienced by the system, possible VM rejections caused by inefficient use of resources, or even increase in power consumption.

3.3 Consolidation strategies

Common consolidation approaches can be enumerated as follows:

- **Incremental release using thresholds:** Using this strategy, the PM decides to migrate its VMs incrementally over time with the hope of full release after a while. The re-consolidation process is triggered when the utilization of the PM fall below the threshold. Setting the threshold in this approach is inevitable, since the number of migrations should be limited in some way.

- **One-shot merge:** The PM decides to migrate its VMs when another PM, with sufficient capacity can be found, to accommodate all its VMs. Since the decisions are based on the state of the PMs involved, there is no need for any configuration on thresholds.

- **One-shot merge + dimension aware re-distribution:** In both above-mentioned scenarios, it is assumed that accommodating VMs in fewer machines is the only factor affecting the efficient consolidation. The intuition is that the resources should either be fully utilized or be in the idle mode. However, they have ignored the fact that the requested resources have shapes and the way these shapes are put together would also affect the efficiency of utilization.

  As it is depicted in Figure 1, efficient distribution of VMs over the PMs can lead to the probability of better consolidation and thus lower power consumption. We define the efficient distribution of VMs as the one in which both CPU and memory in each PM are used in a balanced proportion. We propose a P2P gossip-based protocol which takes into account both, the possibility of releasing a PM and re-distributing the VMs among the peers, to balance the consumption of both dimensions on the each PM.

3.4 Dimension aware consolidation protocol:

In this section, we introduce a gossip protocol for VM placement and consolidation. The protocol is an iterative algorithm, starting from an arbitrary initial

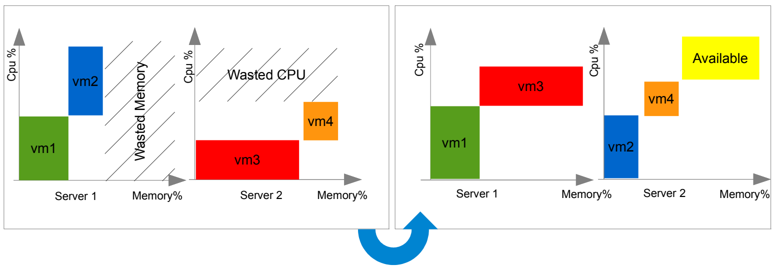


Figure 1: Redistribution of VMs can increase the balance between memory and CPU and improve the consolidation

VM placement. In this protocol, each pair of neighboring peers exchanges gossip messages. This state exchange triggers a local decision function that can possibly lead to a local state change in the associated peers. The decision function evaluates the states of the pair with respect to a common local objective and it proposes a new distribution of VMs between the associated peers that maximizes this objective considering the migration costs. In another word, the protocol continuously moves toward the optimum by iteratively applying a control operator to the peers' states and substitutes their states if the new state has a greater value. Each peer keeps track of the list of VMs assigned to them by the decision function and considers the new list as their updated state for future interactions. The protocol continues the process until it converges and no more VM is re-distributed. When the protocol is converged, a reconfiguration plan can be devised to migrate the VMs from their original location to their assigned peer. The pseudo-code of gossip interactions is illustrated in Algorithm 1.

3.4.1 Decision function

The state exchange between the neighboring peers triggers a decision function. Using this function, the peers involved in the interaction, assesses if the redistribution of their VMs can lead to a better consolidation. The function evaluates the possibility of re-consolidation for either of the following cases:

1. If the aggregated load of the pair, involved in the negotiation, can be accommodated in one peer, then the VMs would be deployed in one peer and the released peer is either set into the power saving mode or considered as free space to be used for other purposes in the future.

Algorithm 1 Gossip Protocol

```
procedure ACTIVE THREAD
    loop
        wait $\Delta$
        for Each neighbor k in the local view do
            Send *myState*
            Receive the state from neighbor k, $State_k$
            *newState*=Decisionfunction()
            *myState*=Update(*newState*)
        end for
    end loop
end procedure
────────────────────────────────
procedure PASSIVE THREAD
    loop
        Receive $State_i$ from i
        Send *myState* to i
        *newState*=Decisionfunction()
        *myState*=Update(*newState*)
    end loop
end procedure
```

2. If not, the function assesses whether re-distribution of the VMs can lead to a more balance utilization of CPU and memory per PM, for both peers.

If any of the above-mentioned conditions are met, the function proposes a new re-distributed set of VMs for each peer, based on the following steps:

- **Step 1: Calculate the Imbalance factor**

  The algorithm calculates the *imbalance* ratio for the possible permutations of VMs, using Equation (5). If re-distributing the VMs leads to the possible future state of (S'_1, S'_2) for $peer_1$ and $peer_2$ and imposes a CPU and memory consumption of (c_1, m_1) and (c_2, m_2) at time *t'*, respectively, the *imbalance* ratio for this specific VM distribution is calculated as:

$$Imb(S'_1, S'_2) = \phi(S'_1, S'_2) \times (\sum_{i=1}^{2} |c_i - m_i|) \tag{5}$$

  Where $\phi(S'_1, S'_2)$=

$$\begin{cases} 0 & \text{if } (c_1 + c_2 \leq \alpha C_{PM}) \ \ \& \ \ (m_1 + m_2 \leq \alpha M_{PM}) \\ 1 & \text{if } (c_1 + c_2 > \alpha C_{PM}) \ \ \& \ \ (m_1 + m_2 > \alpha M_{PM}) \end{cases} \tag{6}$$

- **Step 2: Calculate the migration costs**
  Since the migration of VMs is an expensive task, the algorithm considers the cost of migration when deciding on re-distributing the VMs. The cost of migration is defined as a function of migration time. The total migration time of a VM is calculated as:

$$t = t_i + t_c + t_s + t_r \tag{7}$$

  where t_i denotes the time for iteratively transferring the memory pages, t_c is the time for suspending the VM at source, t_s is the time for CPU transfer and t_r is the time for resuming the VM destination. In the above equation, t_c, t_s and t_r are rather short, however the iterative memory transfer is hard to predict and depends on the memory consumption of the VM.

- **Step 3: Select the VM sets that maximize the *Gain***
  Finally, the algorithm selects a distribution set which leads to a better consolidation among the peers with minimum cost of migration. In the other word, the algorithm selects a distribution of VMs in which the peers can gain the most from transitioning from their current states (*S₁*, *S₂*) to a transferred state (*S'₁*, *S'₂*) with the minimum migration cost. The gain is quantified via Equation (8).

$$Gain() = \frac{Imb(S_1, S_2) - Imb(S_1' - S_2')}{Migration\ cost} \tag{8}$$

The general steps of the algorithm are illustrated in Algorithm 2. It should be noted that finding a VM distribution, defined by CPU and memory, is a 2D bin packing problem and the complexity of the search space grows with the number of VMs deployed in each peer.

3.5 Reconfiguration Plan

In previous section, a mapping between the VMs and the PMs in the system is devised, aiming for efficiency of resource utilization and low power consumption. It has also taken into account the cost of migration in terms of the required memory transfer. However, these VMs are usually serving real time requests or in some cases they are associated with stateful data on the PMs. In these cases, the cost of migration is not just the cost of memory transfer. Therefore, a re-configuration plan should be devised to carefully consider the real time factors such as VMs states, the available network bandwidth and the durability of re-configurations.

Algorithm 2 Decision Function

```
Require: MyState [$c_{me}(t)$, $m_{me}(t)$, list of $VM_{me}$] , NeighborStates [$c_k(t)$, $m_k(t)$, list of $VM_k$]
  if $(c_{me} > \alpha \times C_{PM}) or (c_{me} > \alpha \times M_{PM})$ then
    Offload();
  else if $(c_{me}(t) + c_k(t) \leq \alpha \times C_{PM})$ & $(m_{me}(t) + m_k(t) \leq \alpha \times M_{PM}))$ then
    if $(m_{me}(t)$ ¡ $m_k(t))$ then
      Add the references of VMs on $me$ to $VM_k$.
      Tag $me$ as to-be-idled.
    else
      Add the references of VMs on $k$ to $VM_{me}$.
      Tag $k$ as to-be-idled.
    end if
  else if $((c_{me}(t) + c_k(t) \geq C_{PM})$ & $(m_{me}(t) + m_k(t) \geq M_{PM}))$ then
    Select a distribution of VMs that minimizes the total *Imbalance* for each PM in both peers with minimum memory transfer.
  end if
```

3.6 Offload

As mentioned earlier, in an environment with changing demand, having overloaded PMs is inevitable. In this situations, the system should decide on how to offload the overloaded PM. When a PM becomes overloaded, it contacts its neighbors to find a suitable PM with sufficient capacity to offload its load. On the first round, the peer tries to select among the active neighbors and see if they can accommodate the extra load. In each interaction, the algorithm examines a subset of VMs on the overloaded PM in which it can be accommodated in the neighboring peer with minimum migration cost. The released capacity after migrating this set should be sufficient enough to offload the peer to fall below the overload bar. This subset is devised via an exhaustive search among the VMs currently deployed on the PM.

If none of the active neighbors have sufficient capacity for the offload, the peer activates one of its idle neighbors, if it has any, or it waits for the next cycle to receive a new set of random neighbors via the peer sampling service and repeats the above procedure.

4 Evaluation

We evaluate the performance of three consolidation strategies, discussed in Section 3, and we compare their performance with a random VM placement as the benchmark. The investigated strategies are:

Algorithm 3 Offload

```
for Each neighbor k in the local view do
    if (k is active) & $c_k(t) < Overload\ bar$) & ($m_k(t) < Overload\ bar$) then
        List all the possible subsets of myVMs as the possible sets to be transferred
    end if
    for Each VM subset do
        Calculate the total CPU and Memory demand of each VM subset as the CPU and memory to be transferred
        if (($C_k - c_k(t)$) $\geq transfer\ Cpu$) & (($M_k - m_k(t)$) $\geq transfer\ Mem$) & ($c_{me}(t) - transfer\ CPU < Overload\ bar$) & ($m_{me}(t) - transfer\ Mem < Overload\ bar$) & ($transfer\ Mem\ is\ minimum$) then
            Select the subset
            overload-Resolved=true;
        end if
    end for
    if overload-Resolved==true then
        Break
    end if
end for
if subset!= null then
    Transfer the subset to the neighbor
end if
if subset= null then
    Activate one the idle PM
    Transfer the extra load
end if
```

1. Random VM placement
2. Incremental consolidation using thresholds
3. One-Shot Merge
4. One-Shot Merge + re-distribution w.r.t *Imbalance* ratio

4.1 Performance metrics

The performance of each strategy is evaluated with respect to the following metrics:

1. Data center power consumption
2. Number of active PMs

3. Imbalance rate: It is calculated as:

$$Imb_{dc}(t) = (\sum_{i=1}^{n} |c_i(t) - m_i(t)|)/n \tag{9}$$

where *$Imb_{dc}(t)$* is the average of data center's imbalance at time t, n is the total number of PMs, both idle and non-idle, *$c_i(t)$* and *$m_i(t)$* is the CPU and memory utilization of *PM_i* at time t.

4. Average resource utilization: Resource utilization has a direct impact on power consumption. It is defined as the average utilization of non-idle PMs over each dimension, i.e. CPU and Memory. Hence, for each type of resource, the average resource utilization is calculated as:

$$U_{dc}(t) = \frac{\sum_{j=1}^{m} Res_j(t)}{n_{active} \times Capacity_{PM}} \tag{10}$$

Where m is the number of placement requests at time t, *$Res_j(t)$* is the utilized capacity, either CPU or memory, by *VM_j* at time t, n_{active} is the total number of non-idle PMs, *$Capacity_{PM}$* is the capacity of each PM.

5. Computation time: The computation time is the time it takes for the protocol to compute an efficient VM to PM mapping and it is measured in terms of number of cycles it takes for the protocol to converge.

6. Number of migrations

7. Overload time: The aggregated timestep that the system faces overload.

8. Number of rejected VM requests

4.2 Simulation setup

The evaluation is performed through simulation of a data center in PeerSim [11]. Our data center consists of 100,000 PMs, each has the capacity of 58 vCPU and 64 GB of memory. Our data center offers 6 VM types each fit to a specific use case, similar to Amazon EC2 use cases. The details on the VM characteristics is illustrated in Table 1.

The data center receives 200,000 VM requests during the simulation time. The requests types are uniformly distributed among *memory optimized*, *compute optimized* and *general purpose* VM types. They also intend to run a combination of batch jobs and stateless interactive applications. The interactive applications have long lifetimes, whole simulation run, and their CPU and memory demand is changing over time according to Equation (11), derived from an analysis

Table 1: *VM types and their capacity details*

Category	**VM name**	**vCPU**	**Memory**	**vCpu%**	**Mem%**
Compute	c1.medium	15	7.44	13%	6%
Compute	c3.xlarge	30	14.88	27%	12%
Memory	m2.xlarge	6.5	17.1	6%	13%
Memory	m2.2xlarge	13	34.2	12%	27%
General	m3.medium	3.24	3.75	3%	3%
General	m3.large	6.5	7.5	6%	6%

on google traces introduced in [12]. The number of interactive applications is constant during the simulation run.

$$CPUDemand_j(t) = (\frac{CPU_l}{2})(1 + u_m sin(\frac{2\pi t}{86400} - 2\pi s_m) \tag{11}$$

$$MemDemand_j(t) = \beta \times CPUDemand_j(t) \tag{12}$$

where CPU_l is the expected maximum CPU demand of VM type l, u_m, $s_m \in$ [0,1] is selected uniformly at random. β is the CPU/memory capacity ratio of the VM type l, e.g. $\beta = 0.5$ for a compute optimized VM.

The second group of VMs are a set of batch jobs with the constant CPU and memory demand. The arrival rate of these requests follows a Poisson distribution with $\lambda = \frac{SimulationTime}{2}$, and they have a lifetime follows a truncated power-law distribution with exponent 2, truncated to the length of the simulation run.

We set the utilization factor $\alpha = 0.9$, to ensures the tolerance of the system to performance degradations caused by resource contentions between neighboring VMs. We also assume a PM power consumption in the idle state is 175W and 250W when fully utilized. Each peer maintains a local view of k=20 neighbors which is being updated by the peer sampling service in each cycle. The simulation time is 1000 timesteps and the monitoring system samples the VM demands each 20th timestep. During the two consecutive monitoring, the system load is considered constant. All results presented are the average of 10 simulation runs with the same configuration.

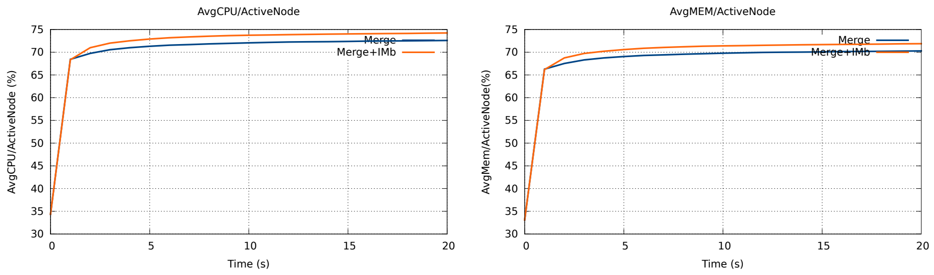
5 Results and Discussions

5.1 Feasibility of P2P approach

In this section, we investigate the feasibility of having a P2P protocol to perform VM consolidation. The main concern when designing a resource manager for a large scale distributed system is the cost of the decision making, usually defined in terms of computation time and the bandwidth consumption. We investigate the cost of our P2P process, in terms of convergence cycles representing the computation time and network bandwidth consumed by gossip protocol during the decision process.

5.1.1 Convergence cycle

In the first experiment, we investigate the time required for the algorithm to reach a stable VM-PM mapping for a specific load. This time is defined in terms of the number of cycles it takes for the protocol to converge. To do this, we considered a time interval in the experiment in which the load is not changing. Hence, the number of the VMs and the total CPU and memory demand during this interval is completely constant.



(a) Convergence on CPU utilization *(b)* Convergence on memory utilization

Figure 2: Computation time in terms of convergence cycles

Figure 2 shows the trend of the average CPU and memory utilization over the time interval with the constant load for the proposed P2P protocols with *Merge* and *Merge+Imb* strategies. The graph shows a fast convergence of the P2P protocols as it reaches a high utilization within 2 to 3 cycles, and it completely converges at 7th cycle, for the configuration specified for this experiment.

5.1.2 Bandwidth consumption

For each consolidation decision, the maximum bandwidth consumed by the gossip protocol, to reach a VM-PM mapping, is calculated by the following equation:

$$\begin{aligned} Bandwidth\ consumption = {} & Number\ of\ PMs \times convergence\ cycle \\ & \times Number\ of\ exchanged\ messages \\ & \times message\ size\ (Byte) \end{aligned} \tag{13}$$

The maximum bandwidth consumption required for a consolidation decision in a data center with 100,000 PMs is $160MB = 100,000 \times 7 \times 2 \times 120$ byte. However, it should be emphasized that this is the maximum bandwidth consumption since in each cycle, a number of peers are excluded from the peers who exchange messages. Messages contain metadata about the type, CPU and memory consumption of each VM, resides on each peer.

5.2 Decision strategy

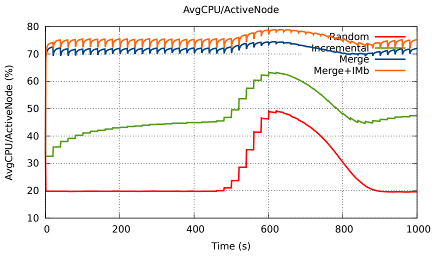
Figure 4 shows a comparison on different performance metrics when each of the strategies are adopted. As it can be seen in Figures 3a, 3b and 3c, after the random placement, *Incremental* strategy on average has the lowest resource utilization, and the highest number of active PMs. It is because, by setting a threshold for triggering the re-consolidation process, some of the possible load consolidations are automatically ignored. Hence, it results in higher number of active PMs and higher power consumption.

The comparison between *Merge* and *Merge+Imb*, shown in Figures 3a, 3b, 3c and 3d, shows higher CPU and memory utilization and lower number of active PMs and lower power consumption for the latter. However, as it can be seen in Figure 3e, this result comes with the price of higher number of migrations. These extra migrations are performed to reduce the imbalance between CPU and memory utilization in each PM.

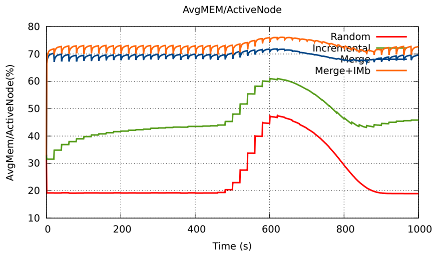
The better performance in *Merge+Imb* strategy is because of accounting the dimensionality of the resources while re-distributing the VMs. The *Merge+Imb* strategy tries to reduce the imbalance between the CPU and memory utilization, depicted in Figure 4a. A balanced utilization of a two dimensional resource results in more efficient utilization of resources in both dimensions, thus lower rejections of VM requests, as shown in Figure 4b, and faster resolution of overloaded PMs, as shown in Figure 4c. By re-distributing the VMs to achieve a lower imbalance, we can have 28% lower rejections, 12.6 % faster offload, for a higher number of overloaded PMs and also 3.2 % lower power consumption for the cost of 25% more migration.

5.3 Impact of local view size (number of neighbors)

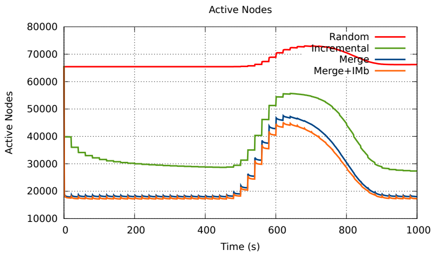
We investigate impact of the local view size on the protocols performance. In this experiment, we consider the data center with 50,000 PMs and 100,000 VM requests and a static load. We vary size of the local view to K=10, 20, 30, 50. Results, shown in Figure 5, indicate that the larger the size of the local view, the faster the protocol can converge. However, this impact become less and less significant for the sufficiently large local views.



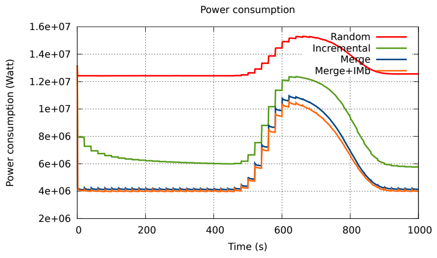
(a) Average CPU utilization



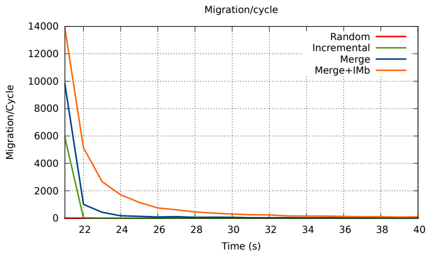
(b) Average memory utilization



(c) Number of active PMs

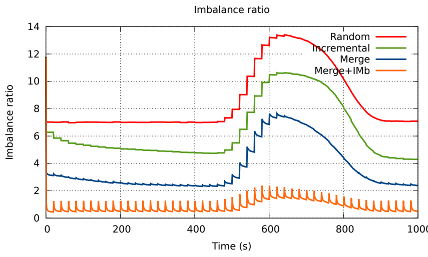


(d) Power consumption

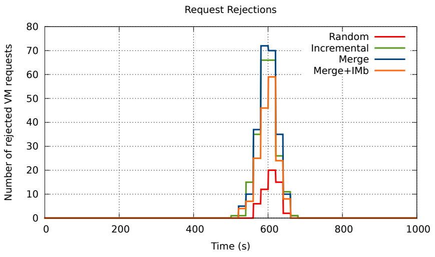


(e) Number of migrations during a constant load interval

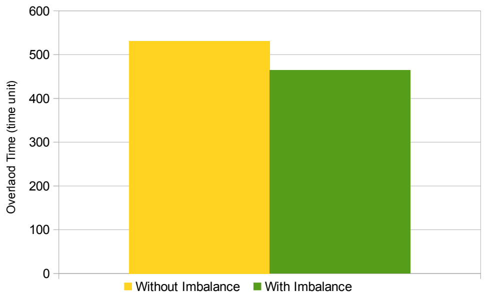
Figure 3: Comparison between different decision strategies



(a) Imbalance rate

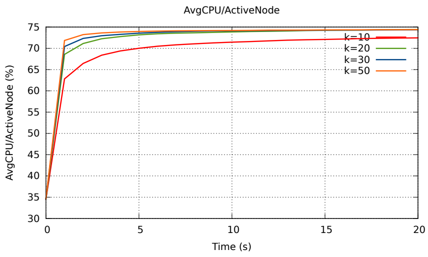


(b) Batch rejections

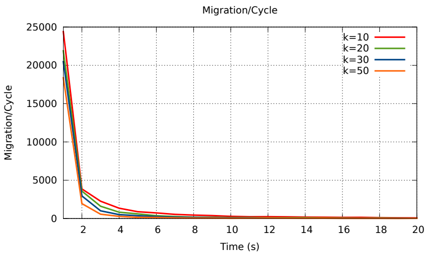


(c) Overload time

Figure 4: Comparison between different decision strategies



(a) Impact of local view size on convergence of average CPU)



(b) Impact of local view size on the number of migrations

Figure 5: Impact of local view size

6 Related work

The problem of VM consolidation is discussed in different studies. The following is a brief outline:

Beloglazov and his colleagues in [13] used a *best-fit* heuristic for initial VM placements and further on, they migrate the VMs if a violation on one of the upper or lower utilization thresholds is occurred. However, they based their placement decisions on only one dimension, CPU consumption, and they also ignored the cases that changing the arrangements of VMs the PMs can lead to a more optimal utilization of resources. The algorithm is centralized and it is examined for a data center with 100 PMs.

Svärd et al. [14] studied a set of heuristics to maintain the optimality of allocations via a set of actions, such as VM and PM suspend and resume, and also VM migration. These actions are triggered when either a PM crashes, or a VM arrives or exits. Their approach has a centralized design and supports up to 48 PMs. We, on the other hand, are interested in solving the problem for larger scale.

Marzolla et al. [15] presented V-Man, a decentralized algorithm, using gossip protocol, for VM consolidation. They modeled the PMs and VM requests as one dimensional entities, and they assumed that applications have constant load. Their algorithm is robust to PM and service failures. However, the model is a bit simplistic and does not cover the complexities of a multi-dimensional placement problem and the dynamic load. They also didn't consider the migration costs while deciding on which VM to migrate.

Wuhib in [16] proposed a resource allocation architecture and a gossip protocol to address a set of well known management objectives, such as fairness, balanced load, energy efficiency and service differentiation. In their model, machines are associated with CPU, memory, and network interface capacity. Their protocol attempts to either minimize the overload if the PM is overloaded, or to minimize the objective function under the constraint of live migration. The results shows that the protocol is effective and scales well, despite the fact that they did not consider the proportionality of resource usage per PM.

Mastroianni and his colleagues [17] proposed a probabilistic consolidation of VMs in a data center. The main idea is that a single PM is the one to decide whether they should accept or reject a VM. When a VM request arrives, the request is broadcasted by a coordinator to all the PMs and they respond the coordinator if they can accept the request. This decision is based on a Bernoulli trial, which ensures that the PMs tend to respond positively when they have intermediate utilization values for both CPU and memory. When the local decisions are made, a data center manager coordinates the decisions. Finally, the coordinator selects one of the respondents randomly. A PM also decides to migrate its VMs if it is too low utilized or high utilized, in the similar fashion.

A consolidation algorithm is proposed in [7], focusing on the trade-offs between energy consumption, performance and resource utilization. They argued that consolidation leads to performance degradations and longer execution times, therefore, to save the energy an optimal consolidation rate should be carefully determined. The intuition is similar to our approach on measuring *imbalance* rate for allocation and it is to ensure that both dimensions are equally used. However, they also discussed the problem for the one time consolidation without considering that the load of the environment is changing and the initial consolidation and also the optimal point is susceptible to change.

7 Conclusion

In this paper, we investigated the potentials and advantages of P2P systems for making complex resource management decisions. We discussed the common design concerns regarding VM consolidation and gave a brief comparison among them. We formulated the consolidation problem in a P2P fashion, to achieve scalability and support complex decisions in a short computational time. We also presented a P2P gossip-based protocol for multi-dimensional VM placement and consolidation, considering the changes in both VMs demand and infrastructure load.

Through extensive experiments, the results show that the P2P approach is feasible and scalable up to 100,000 PMs and 200,000 VM requests. It also produces, resource utilization of 75%, on both dimensions, CPU and memory, when the consolidation aim is 90%. This result is produced within a short computation time, less than 7 cycles, for the examined scale, which is essential to be responsive in a dynamic environment. Based on these results, we can argue that dividing management responsibilities to a set of identical autonomic elements allows the system to scale without compromising the complexity of the problem or quality of the solution, that is required to keep the response time within an acceptable time frame. Adopting a P2P approach, also eliminates cumbersome configuration settings that are required in the centralized approaches.

The observations also indicate that a balanced utilization of a two dimensional resource, in a mixed workload, results in more efficient utilization of resources in both dimensions. This leads to a lower rejections of future VM requests and faster resolution of overloaded PMs. The results shows 28% lower rejections, 12.6% faster offload, for a higher number of overloaded PMs and also 3.2% lower power consumption for the cost of 25% more migrations.

8 Acknowlegement

Financial support for this work is provided in part by the Swedish Research Council (VR) under contract number C0590801 for the project Cloud Control and the Swedish Government's strategic research project eSSENCE.

References

[1] A. Corradi, M. Fanelli, and L. Foschini, "VM consolidation: A real case based on openstack cloud," *Future Generation Computer Systems*, vol. 32, pp. 118–127, 2014.

[2] M. Sedaghat, F. Hernandez, and E. Elmroth, "Autonomic resource allocation for cloud data centers: A peer to peer approach.," in *The ACM Cloud and Autonomic Computing Conference (CAC'14), Accepted*, 2014.

[3] L. Minas and B. Ellison, "The problem of power consumption in servers," *Intel Corporation. Dr. Dobb's*, 2009.

[4] S. Lee, R. Panigrahy, V. Prabhakaran, V. Ramasubramanian, K. Talwar, L. Uyeda, and U. Wieder, "Validating heuristics for virtual machines consolidation," *Microsoft Research, MSRTR-2011-9*, 2011.

[5] S. He, L. Guo, M. Ghanem, and Y. Guo, "Improving resource utilisation in the cloud environment using multivariate probabilistic models," in *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pp. 574–581, IEEE, 2012.

[6] M. Jelasity and M. Van Steen, "Large-scale newscast computing on the internet," tech. rep., Citeseer, 2002.

[7] S. Srikantaiah, A. Kansal, and F. Zhao, "Energy aware consolidation for cloud computing," in *Proceedings of the 2008 conference on Power aware computing and systems*, vol. 10, USENIX Association, 2008.

[8] J. Rao, Y. Wei, J. Gong, and C.-Z. Xu, "Dynaqos: model-free self-tuning fuzzy control of virtualized resources for qos provisioning," in *Quality of Service (IWQoS), 2011 IEEE 19th International Workshop on*, pp. 1–9, IEEE, 2011.

[9] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes, "Cloudscale: elastic resource scaling for multi-tenant cloud systems," in *Proceedings of the 2nd ACM Symposium on Cloud Computing*, ACM, 2011.

[10] C. Klein, M. Maggio, K.-E. Årzén, and F. Hernández-Rodriguez, "Brownout: building more robust cloud applications.," in *ICSE*, pp. 700–711, 2014.

[11] A. Montresor and M. Jelasity, "Peersim: A scalable p2p simulator," in *Peer-to-Peer Computing, 2009. P2P'09. IEEE Ninth International Conference on*, pp. 99–100, IEEE, 2009.

[12] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Heterogeneity and dynamicity of clouds at scale: Google trace analysis," in *Proceedings of the Third ACM Symposium on Cloud Computing*, ACM, 2012.

[13] A. Beloglazov, J. Abawajy, and R. Buyya, "Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing," *Future Generation Computer Systems*, vol. 28, no. 5, pp. 755–768, 2012.

[14] P. Svärd, W. Li, E. Wadbro, J. Tordsson, and E. Elmroth, "Continuous datacenter consolidation," 2014.

[15] M. Marzolla, O. Babaoglu, and F. Panzieri, "Server consolidation in clouds through gossiping," in *World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2011 IEEE International Symposium on a*, pp. 1–6, IEEE, 2011.

[16] F. Wuhib, R. Yanggratoke, and R. Stadler, "Allocating compute and network resources under management objectives in large-scale clouds," *Journal of Network and Systems Management*, pp. 1–26, 2013.

[17] C. Mastroianni, M. Meo, and G. Papuzzo, "Probabilistic consolidation of virtual machines in self-organizing cloud data centers," *IEEE Transactions on Cloud Computing*, vol. 1, no. 2, pp. 215–228, 2013.

Paper IV

Decentralized Cloud Datacenter Reconsolidation through Emergent and Topology-aware Behaviour

M. Sedaghat, F. Hernandez-Rodriguez, and E. Elmroth

Future Generation of Computer System, Elsevier, Vol. 56, pp. 51-63, 2016.

Decentralized Cloud Datacenter Reconsolidation through Emergent and Topology-aware Behaviour[*]

Mina Sedaghat[†], Francisco Hernandez-Rodriguez[†], Erik Elmroth[†]

Abstract

Consolidation of multiple applications on a single Physical Machine (PM) within a cloud data center can increase utilization, minimize energy consumption, and reduce operational costs. However, these benefits comes at the cost of increasing the complexity of the scheduling problem.

In this paper, we present a topology-aware resource management framework. As part of this framework, we introduce a Reconsolidating PlaceMent scheduler (RPM) that provides and maintains durable allocations with low maintenance costs for data centers with dynamic workloads. We focus on workloads featuring both short-lived batch jobs and latency-sensitive services such as interactive web applications. The scheduler assigns resources to Virtual Machines (VMs) and maintains packing efficiency while taking into account migration costs, topological constraints, and the risk of resource contention, as well as the variability of the background load and its complementarity to the new VM.

We evaluate the model by simulating a data center with over 65000 PMs, structured as a three-level multi-rooted tree topology. We investigate trade-offs between factors that affect the durability and operational cost of maintaining a near-optimal packing. The results show that the proposed scheduler can scale to the number of PMs in the simulation and maintain efficient utilization with low migration costs.

Keywords: Cloud computing; Peer to Peer; VM consolidation; Resource management.

[*]The paper has been re-typeset to match the thesis style.

[†]Dept. of Computing Science, Umeå University, SE-901 87 Umeå, Sweden, email: {mina, francisco, elmroth}@cs.umu.se

1 Introduction

Cloud providers offer an infrastructure to be shared by multiple applications, which is usually expensive and needs to be wisely utilized. Utilization can be improved by running an appropriate mix of application workloads on each individual machine, which is known as consolidation. While consolidation can be used to increase utilization, it also increases the complexity of the scheduling problem [1]. Complexity comes from the fact that application workloads are often heterogeneous with respect to their size, lifetime, performance sensitivity, and the type of resources they use, i.e. whether they are CPU- or memory-intensive. A degree of sub-optimal application placement is inevitable due to load changes and the fact that it would be impractically expensive to completely re-map every component of every running application across all of the available servers each time a load change occurred. Consequently, there is a need for a scheduler that can respond rapidly to changes in demand, producing efficient and durable packing in a way that accounts for the heterogeneity of the cloud's workloads, imposes low costs of maintaining the packing efficiency, and can scale up to tens of thousands of servers per data center. We consider a packing to be durable if it does not necessitate frequent migrations in order to maintain the usage efficiency of allocations.

Consolidation is intrinsically a computationally hard problem. Several groups have formulated consolidation as an Integer Linear Programming (ILP) problem, which can be solved relatively quickly [2–5]. However, the ILP approach does not scale well and becomes unfeasible when dealing with larger data centers and/or more severe packing constraints. To achieve scalability, an ILP formulation must either compromise on the quality of the solution in order to maintain a response time that is within acceptable limits or alternatively impose a static partitioning scheme on the infrastructure, which limits the efficiency of resource utilization because one fixed partition may be underutilized while another is over-utilized [1].

Here, we propose a new P2P consolidation framework. Some of this framework's basic functionality has previously been verified in prototype form [6]. The proposed framework is a general computational model for cooperatively optimizing a global system objective through local interactions and computations in a multi-agent system over a semi-random connectivity. We also introduce a scheduling heuristic designed to provide and maintain durable packing with low maintenance costs for a data center with a dynamic workload. A scheduler based on this heuristic is shown to achieve such durable packing in a way that avoids costly reconfigurations, and to offer cheap migration plans (i.e. schemes that specify which workloads should be migrated to which resources) to maintain packing efficiency. The framework's P2P structure provides parallelization with a high degree of concurrency, and also helps to minimize the time required for computations while improving scalability. The random overlay used in the proposed P2P structure allows the system to create a logical dynamic connectivity among a large pool of resources, dynamic cells, and reduces the negative impacts of static partitioning (which can lead to low utilization). Formulating the consolidation as a distributed optimization problem allows the system to factor in more sophisticated trade-offs than are possible with the ILP approach because it avoids the need for a highly loaded centralized scheduler. For example, the scheduler can determine whether to migrate a VM to a remote Physical Machine (PM) on another cluster or deploy it on a PM that is more nearby but subject to a higher risk of resource contention. The local decision making employed within the P2P framework also reduces the amount of monitoring data that must be collected and transferred over the network.

The main contributions of this paper are:

- A formulation of the VM consolidation problem as a distributed optimization problem.
- A topology-aware resource management framework for VM consolidation.
- A heuristic algorithm for VM consolidation that factors in the risks of resource contention, packing efficiency, migration costs, and migration locality to produce durable consolidations and offer cheap migration plans to maintain packing efficiency and reduce resource stranding.
- An in-depth simulation-based evaluation of the system behavior under different settings and configurations. The results obtained in this evaluation show that the proposed scheduling framework can produce durable consolidations for large numbers of VM requests with varying demands, arriving over a simulation time of 24 hours at a data center with over 65000 PMs. The framework scales to the tested number of PMs and maintains efficient resource utilization with low migration costs.

The remainder of the paper is organized as follows. In Section 2, we discuss the requirements and challenges of scheduling a mixed workload. Section 3 presents problem statements. In Section 4, we present the full P2P framework and the proposed heuristic algorithm. Section 5 describes the experimental setup, and Section 6 reports the evaluation and analysis of the results. Finally, we discuss related works and offer some concluding remarks in Sections 7 and 8, respectively.

2 Challenges and Requirements

When scheduling VMs to run different services or batch jobs, the scheduler must meet several requirements and it faces a number of challenges in meeting them. A summary of these challenges are:

1. **Resource contention caused by consolidation**: Co-locating different applications can cause performance variability or degradation due to resource contention when resources are being shared [7]. The scheduler should therefore identify complementary workloads and place them together to improve packing efficiency and minimize resource contention.

2. **Job heterogeneity**: A data center will be required to run different types of applications. In broad terms, two classes of application can be distinguished: long-running interactive services and batch jobs, which perform a specific computation and then finish. Batch jobs that are run in cloud data centers are usually shorter and less latency-sensitive than interactive services, involve constant resource utilization, and do not usually require careful scheduling [1, 8]. It would thus be best to devote most of the scheduler's available time and resources to the placing of interactive services and to spend relatively little time on scheduling batch jobs [1]. In addition, some tactics that can be applied to batch jobs in order to reduce the burden on an overloaded server could not acceptably be applied to interactive jobs. For example, a batch job could be stopped and restarted later, or the VM on which it is running could be transferred to another server via a cold migration. Neither approach would be possible for a VM running a latency-sensitive interactive service. Jobs can be further distinguished on the basis of other characteristics such as their lifetime, size, and performance sensitivity in order to develop effective strategies for fixing sub-optimal allocations that lead to the over- or under-loading of individual servers.

3. **Migration cost**: VM migration is a widely used technique for achieving consolidation once the decision on which jobs to consolidate has been made [9]. However, migrations are often costly. Particularly important costs to consider include the cost of double resource utilization during the migration, the costs of SLA violations caused by migration downtime, the cost of network traffic, and the potential network contention issues that may arise during the migration. The scheduler should produce a cheap migration plan with a minimal impact on the performance of the running applications. A migration plan may specify which type of migration is

to be performed (cold or live), a candidate destination PM, and a list of VMs (selected based on their migration costs) to be migrated.

4. **Topological constraints**: The scheduler should consider the network topology to avoid high migration costs due to network traffic, contentions, or redundant configurations. Most existing works on scheduling treat the data center as an unstructured pool of resources, but real data centers Virtual LANs (VLANs), Access Control Lists (ACLs), broadcast domains, and load balancers that impose constraints and create barriers that reduce the scope for agility in migration [10].
5. **Risk of load change and contention**: The scheduler should factor the risk of change and contention into its decision function so as to avoid frequent migrations and produce durable decisions.
6. **Computation time**: The scheduler should produce a solution within an acceptable time-frame, and before the solution becomes disparaged due to load changes.

3 System model

We define the problem of VM consolidation as an optimization problem. In our model, the data center has n PMs. Each PM has a CPU, memory and bandwidth capacity, C_{PM}, M_{PM} and B_{PM}, respectively. The CPU, memory and bandwidth utilization of PM i at time t are denoted as $c_i(t)$, $m_i(t)$ and $b_i(t)$, respectively. For clarity, it is assumed that all PMs are homogeneous, although the formulation can also be generalized to heterogeneous machines.

PMs in the data center are organized in a three-level multi-rooted tree structure, as shown in Figure 1. PMs are the leaves of the network tree and they are linked to edge switches, which are further connected to aggregation switches and finally to the core switches. The data center consists of a core level with 1 core router, an aggregator level with 2 clusters, a group level with 32 groups, and the physical machine level with 1024 physical machines.

The data center offers l VM types, each of which has an expected compute, memory and bandwidth capacity of CPU_l, Mem_l and $Bwdth_l$. These VM types are grouped into three different categories - *compute optimized*, *memory optimized*, and *general purpose* - on the basis of their dominant characteristics.

It is assumed that m VM placement requests reach the system during the data center's operating time. Each request j demands a specific VM type l and consumes a given $CPUDemand_j(t)$, $MemDemand_j(t)$ and $BwdthDemand_j(t)$ at each timestep, t. Depending on the type of the application, each VM has a page dirty rate of R_j (kb/s).

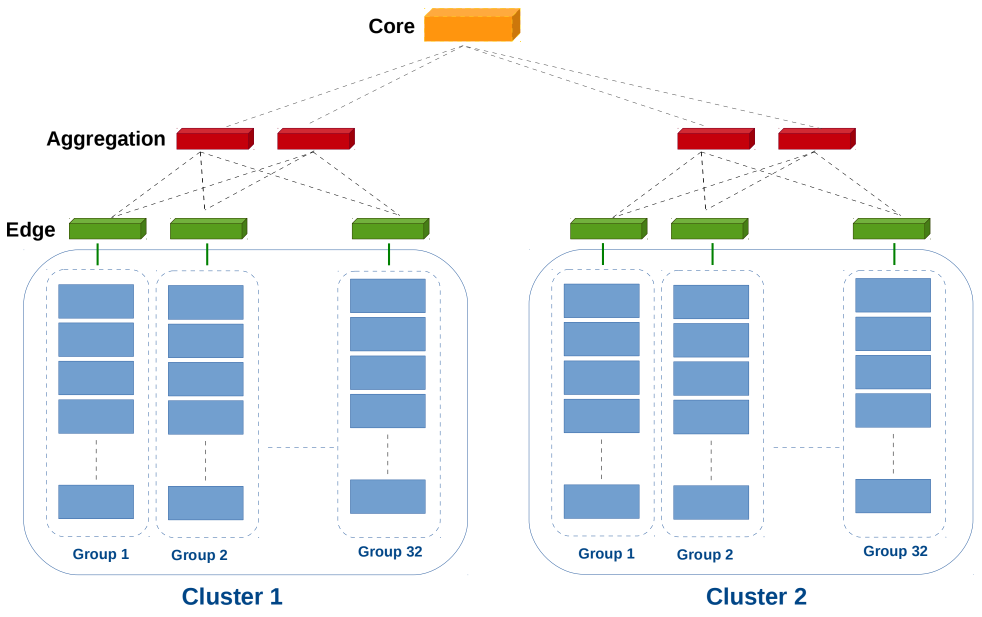


Figure 1: Data center topology

The data center uses the pre-copy approach for live VM migration, and migrations are assumed to be performed in-band, i.e. the same bandwidth is used by the migration process and the service running in the VM such that $Bwdth_{mig} = \phi * Bwdth_l$, where $\phi < 1$. There is no shared storage for VM migration and the VMs' data is migrated fully from the source to the destination PM.

The overall optimization problem for the data center's scheduling is formulated in terms of maximizing the data center utility $U_{dc}(t)$:

$$Maximizing\ U_{dc}(t) = \frac{\sum_{j=1}^{m} Res_j(t)}{n_{active} \times Capacity_{PM}}, \tag{1}$$

Where m is the number of placement requests at time t, $Res_j(t)$ is the capacity utilized by $VM\ request_j$ at time t, and n_{active} is the number of non-idle PMs. α is a utilization factor, defined to avoid performance degradation caused by interference among the consolidated VMs, and $Capacity_{PM}$ is the capacity of the PM in terms of CPU and memory. If the system features heterogeneous PMs, the constant $Capacity_{PM}$ should be replaced by the capacity for each individual type of PM.

$$\sum_{j=1}^{m} Res_j(t) \leq n_{active} \times (\alpha \times Capacity_{PM}). \tag{2}$$

Achieving efficient utilization is important, but it should not be the sole objective as it does not reflect the operational costs of maintaining the desired packing efficiency, such as migration or network costs, in a dynamic environment over time. Moreover, aggregate utilization hides the negative effects of fragmentation and stranding since CPU and memory utilization are computed independently [8]. High fragmentation and stranding lead to low utilization and rejection of VM requests, even when the total aggregate capacity is sufficient. In addition to issues of resource fragmentation and stranding, placement becomes sub-optimal because of load changes and the frequent arrival and termination of VMs. However, make-overs to fix sub-optimal placement are very expensive and should either be avoided or planned carefully.

4 Proposed method

We propose a P2P framework, some of whose basic functionality has previously been verified in [6]. Within the framework, PMs are structured as peers, which are connected to form neighborhoods. Each peer is associated with an autonomous *node agent* that monitors the PM's state and makes local decisions based on its local view and policies. Each *node agent* uses a risk-aware, topology-aware heuristic function to discover resources for new VM requests or to improve a sufficiently sub-optimal state, such as an overload or an underload. The *node agents* constantly apply a relatively simple and restricted set of rules in order to improve the packing efficiency in their own neighborhoods, and the overall state of the system is improved by their combined action.

4.1 Logical overlay

Peers are logically connected by an overlay network. The overlay is built and maintained by a peer sampling service known as newscast [11]. The peer sampling service periodically maintains a communication graph by providing each peer with a list of peers to be considered as neighbors. Each peer at each timestep only knows about k random neighbors, shaping its local view. We developed and extended the classical newscast to introduce topological awareness and provide a neighbor list that is based not only on each peer's timestamp, but also on its physical proximity in the data center architecture. A PM has a higher probability of being returned as a sample if it is in the same server group as the main peer. Having a P2P overlay allows us to create logical dynamic connectivity (dynamic partitioning) within a large pool of resources and reduce the negative impacts of static partitioning, which can cause low utilization. However, we consider the physical proximity of the neighbors when building the logical overlay to reduce the costs of network transit and reconfigurations.

4.2 Node agents

Each peer is associated with a *node agent*, which is responsible for functional tasks such as monitoring and resource assignment, and for making consolidation decisions such as placing a new VM request or fixing sub-optimal states (e.g. overloads or underloads). An action event is triggered on the arrival of a new request or when the PM is confronted with a sub-optimal state. The *node agent* selects the best candidate, i.e. that with the lowest score calculated using a heuristic function. The score function is defined in terms of the risk of resource contention, packing efficiency, migration costs, physical proximity and an imbalance value (See Section 4.3.1).

The effectiveness of the scheduling method can be illustrated by the idea of 'Power of 2 random choices' [12]. For each placement request, triggered by an event, the peer sampling service feeds the algorithm with a number of random peers. From these random peers, the algorithm selects the PM that minimizes the score function. This process will then continue evaluating placement options, using new sets of random neighbors in each round, until the predefined time limit for VM requests of the relevant class has been reached; this time limit will be comparatively short for batch VMs and longer for those running services. The placement candidate with the lowest score value will then be selected as the final PM for VM deployment or the destination for VM migration.

The *node agent* consists of the following modules:

- **Monitoring system**: Monitors the resource consumption of the PM and each of the VMs deployed on the PM.
- **PM state profiler**: Uses historical monitoring data to profile the PM's state over a compact sliding time window and creates an *action trigger* if the PM enters a state in which it is *overloaded* or *underloaded* for a sustained period. A PM is considered to be overloaded if its resource (either CPU or memory) utilization exceeds a predefined threshold. A PM is considered underloaded if its resource utilization on both dimensions drops below a predefined threshold. For brevity, we henceforth refer to

both of these states as sub-optimal states. These events are queued, sorted and processed based on their priorities by the *node agent*.

- **VM state profiler**: Uses historical monitoring data on the VM workload to specify the VM state as either memory intensive or compute intensive. This state information is then used by the decision module to co-locate VMs with complementary workloads.
- **Decision module**: Uses a heuristic function to rank neighboring PMs and select the best candidate to resolve the action trigger. It uses the following sub-modules to account for various factors in its decision-making process:
  - Risk calculator: Calculates the risk of placing VM_i on PM_j, taking into account the risk of overloading PM_j and causing resource contention, as well as the risk of demand variability.
  - Distance calculator: Calculates the physical proximity between the associated PM and PM_j based on the data center's topology. The greater the distance, the greater the probability and cost of network contention, and the greater the need for appropriate configuration settings.
  - Data transfer calculator: Predicts the amount of data to be transferred if VM_i is migrated from PM to PM_j. The prediction is based on the migration policy (cold or live), memory utilization, page dirty rate, and the bandwidth available for the migration of VM_i.
  - Efficiency calculator: Calculates the packing efficiency of placement on PM_j with respect to the desired resource utilization level. It should be noted that efficiency can be defined in different ways to support alternative objectives such as load balancing.
  - Imbalance measure: Quantifies the expected relationship between CPU and memory utilization for PM_j if VM_i is placed on it. This is done to avoid the risk of a scenario in which the applications running on a PM fail to fully utilize one or more of its resource types [10]. The objective is to minimize the negative impacts of resource stranding and the risk of inefficient resource usage with respect to any given resource dimension.
- **Migration planner**: Selects a VM set to migrate from a given PM to a selected destination PM_j. The selection is based on the estimated migration cost for each VM and the impact of the migration on resolving the overload.

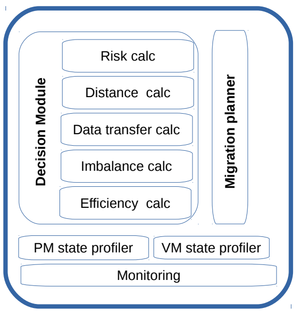


Figure 2: A node agent's internal architecture

Each Node agent is a light weight component that can be plugged into a resource management framework and deployed on each PM (or the scheduling unit).

4.3 Consolidation through resource discovery

The *node agent* initiates a discovery process within its neighborhood that iteratively ranks its neighbors using a score function. Depending on the type of event (new placement request, overload, or underload) and the type of VM request (batch job or service), a computation time (number of iterations) is assigned to produce a placement result.

This time will typically be short for resolving overloads and placement of batch jobs due to their urgency and the fact that they do not require costly planning. Longer times are needed for placing service applications or resolving *underload* events, which require more careful planning. In each iteration, the peer sampling service feeds the *node agent* with a number of new random neighbors and the discovery process continues the search until the allocated time for the request is exhausted. At this time, the PM with the lowest score is selected for placing the VM. We assigned a higher priority to placement of the new VMs than to *server underload* in order to avoid costly migrations by allowing fragmentation problems to be resolved through the deployment of new VM placements.

As shown in Table 1, the events are prioritized based on their urgency and are then processed based on their priorities.

Event	**Priority**
Server overload	1
Arrival of new request	2
Server underload	3

Table 1: *Events are prioritized based on their urgency and then processed based on their priorities*

Algorithm 1 Discovery algorithm

Input: eventList
Output: Best deployment candidate, (bestPeer)

```
1: Sort eventList based on events' priorities
2: while eventList != empty do
3:     Update the list of neighbors
4:     Rank neighbors with available capacity based on their scores.
5:     Select the PM with minimum score, as the best candidate (bestPeer).
6:     if bestPeer!=null and ($event.lifeTime < MaxTTL$) and (eventType = newServiceRequest or eventType = underload) then
7:         Add bestPeer to the candidateList
8:         event.lifeTime ++;
9:         Continue;
10:    else if bestPeer != null and ($event.lifeTime \geq MaxTTL$) or ( eventType = overload or eventType = newBatchRequest ) then
11:        Remove event from peer's eventList;
12:        Perform Placement(bestPeer);
13:        Continue;
14:    else if ($event.lifeTime \geq MaxTTL$) and bestPeer=null then
15:        Remove event from peer's eventList;
16:        Mark the request as pending;
17:        Continue;
18:    else if bestPeer=null then
19:        event.lifeTime ++;
20:        Continue;
21:    end if
22: end while
```

4.3.1 Score function

The *node agent* uses a score function to rank the candidates for VM placement or migration. The score function is a weighted sum of the *risk (Risk)*, *efficiency (E)*, *migration locality (Dist)*, *data transfer cost (DT)* and *imbalance (Imb)* values, normalized against their respective upper bounds. This formulation accommodates a flexible trade-off between the above-mentioned factors through the choice of the $\lambda_i,\ i = 1, \ldots, 5$ values in the following score function:

$$Score = (\lambda_1 \times Risk) + (\lambda_2 \times Dist) + (\lambda_3 \times E) + (\lambda_4 \times DT) + (\lambda_5 \times Imb), \quad (3)$$

where

$$\lambda_1 + \lambda_2 + \lambda_3 + \lambda_4 + \lambda_5 = 1. \quad (4)$$

- **Risk**: We define the risk, $Risk_j$, of selecting PM_j as the product of the risk of resource contention and overbooking, R_o, and the risk of load variation, R_v. The higher the risk value, the less desirable the PM as a deployment candidate.

  $$Risk_j = R_o * R_v \quad (5)$$

  - **Risk of overbooking and contentions**: The R_o value for PM_j when VM_i is going to be deployed is calculated as :

    $$R_o = Max(R_o(CPU), R_o(Memory)) \quad (6)$$

    where $R_o(CPU)$ is the risk of overbooking *CPU* and $R_o(Memory)$ is the risk of overbooking memory when VM_i is deployed. Each of these values [13] is calculated as:

    $$R_o = \begin{cases} 0 & \text{if } Req < Unreq) \\ \frac{Req_i - Unreq_i}{Free - Unreq_i} & \text{if } Unreq \leq Req \leq free) \\ 1 & \text{if } Req < Free \end{cases} \quad (7)$$

    where *Req* is the CPU or memory capacity required by the VM_i, *UnReq* is the difference between the total PM capacity and the capacity requested by the deployed VMs, and *Free* is the difference between total PM capacity and the capacity used by all the deployed VMs.

  - **Risk of load variation**: We define the risk of load variation for PM_j as the Coefficient of Variation (CoV) of the PM_j over the last 10 monitoring steps. CoV captures the relative variation in the workload from its average intensity. PMs with a high CoV are identified as more risky for consolidation [14].

- **Migration distance and locality *(Dist)***: The hierarchical nature of the network results in limited bandwidth availability between the PMs in different clusters because some of the ports (mainly those of layer 3) are usually oversubscribed. The limited bandwidth between PMs of different clusters and the limited agility imposed by network constraints [10] (e.g VLANs, ACLs, broadcast domains, load balancers) makes it desirable to migrate VMs to PMs that are in close physical proximity to the source PM, i.e. belonging to the same server group or cluster. To account for this, we define the *distance* between two PMs as the number of hops a communication message should travel up in the hierarchy to access a PM in another server group or cluster. The cost of migrating a VM between two PMs is lowest when both PMs are in the same server group, and higher if they are members of different clusters. By this definition, the distance between two PMs in a 3 level multi-rooted tree topology will be $1 \leq Dist \leq 3$.

- **Data transfer cost *(DT)***: The data transfer cost is defined in terms of the total traffic generated by the migration; it depends on the migration strategy that is adopted and the amount of data to be transferred. Depending on the type of application running on the VM, we adopt different migration strategies. The first category of applications are batch jobs, which are usually less latency-sensitive than interactive services. Therefore, they can simply be stopped and restarted on another machine, or they can be cold migrated (migrated while they are powered-off). The decision of whether to adopt a stop-restart approach or perform a cold migration depends on the length of the batch job and the state of job completion. In this paper, we assume that all batch jobs are cold-migrated to test the system in an extreme case. However, adopting a stop-restart policy would further reduce the amount of data transfer. The volume of data transfer during a cold migration of a batch job is typically equal to the amount of memory that must be transferred.

  The second category of applications are interactive services, which are latency sensitive and require live migration when being re-located. The data transfer volume for live migrating an interactive service is calculated [15,16] as:

$$DT = M + M \times (\frac{1-(R/L)^{n+1}}{1-R/L}), \tag{8}$$

  where M is the memory size of the VM, R is the VM's page dirtying rate,

L is the bandwidth of the link used for migration and n is the number of pre-cycles, calculated as:

$$n = \min(\log_{(\frac{R}{L})} \frac{T \times L)}{M}), (\log_{(\frac{R}{L})} \frac{(X \times R)}{M \times (L - R)}), \tag{9}$$

where T is the *switch over time setting* and X is the *sufficient progress threshold*, both of which are user settings used in vMotion [17]. They define the time (in milliseconds) required to start the stop-copy process and the point at which sufficient data transfer has been achieved (in MB), respectively. Interactive services are generally poor and unlikely candidates for migration due to their latency sensitivity and the extra data transfer required for the migration of dirty pages during live migration.

- **Efficiency *(E)***: The efficiency of deploying VM_j (CPU_j, mem_j) on PM_i) is defined as the maximum difference between the PM's utilization after deploying VM_j and the desired utilization, over the dimensions of both *CPU* and *memory.*

$$e = Max((Max_{CPU} - (c_i + CPU_j)), \tag{10}$$
$$(Max_{Mem} - (m_i + mem_j))), \tag{11}$$

where Max_{CPU} and Max_{Mem} are the desired utilization of a PM, and c_i and m_i are the CPU and memory utilization of PM_i).

- **Imbalance *(Imb)***: Resource stranding or uneven utilization of resources over different dimensions leads to wastage of resources in one dimension [18–20]. We estimate the *imbalance* rate as:

$$Imb = |(Source_{CPU} - CPU_i) - (Source_{Mem} - Mem_i)| + \tag{12}$$
$$|(Dest_{CPU} + CPU_i) - (Dest_{Mem} + Mem_i)|, \tag{13}$$

where, $Source_{CPU}$ and $Source_{Mem}$ are the CPU and memory utilization of the source PM. CPU_i and Mem_i are the aggregated CPU and memory of the VM or the VM set that need to be migrated, and $Dest_{CPU}$ and $Dest_{Mem}$ are the CPU and memory utilization of the destination PM. When considering deployment of a new request, the imbalance value of the source PM is assumed to be zero. In another words, the imbalance value imposed by a new request is the possible future imbalance if VM_i is deployed on destination PM_j.

5 Experimental Setup

The evaluation of the proposed P2P management framework and the associated heuristics was performed by simulating a large-scale data center using PeerSim [21]. A variable load was applied to the data center due to VM churn and changing VM demand. The simulated data center consists of 65536 PMs interconnected in a multi-rooted tree topology, with 1 core router at the core level, 2 clusters at the aggregator level, 32 groups at the third level, and 1024 physical machines in each group (see Figure 2).

Each PM has a capacity of 16 EC2 Compute Units (ECUs) ($8\ core \times 2\ ECU$) and 64 GB of memory. The data center offers 6 VM types, similar to Amazon EC2 instances [22, 23]. More detailed specifications of the VMs are presented in Table 2.

We ran the simulation for 1440 cycles of one minute each, giving a total simulation time of 24 hours. Each node agent monitored the state of its associated PM every minute and the *PM state profiler* processed the historical monitoring data every 10 minutes to detect potential state triggers. To assess repeatability, we repeated each experiment 10 times, and present calculated Standard Error (SE) values for each metric considered in the analysis.

Category	**VM name**	**ECU**	**Memory(GB)**	**Bandwidth**
Compute	c3.xlarge	4	7.5	62.5
Compute	c3.2xlarge	8	15	125
Memory	r3.large	2	15.25	62.5
Memory	r3.xlarge	4	30.5	125
General	m3.large	2	7.5	62.5
General	m3.Xlarge	4	15	125

Table 2: *VM types and details of their capacities*

5.1 Workload demand

The data center receives 230,000 VM requests during the simulation run, imposing a maximum approximate utilization of 90% if all VMs are accepted and fully utilize their reserved capacity. However, the shapes of the VMs, changes in their demands, and resource fragmentation in data center mean that at any given moment there will be a number of pending requests and so the actual utilization will be below this maximum level. In addition, the reserved capacity for each VM is normally not fully utilized during its life time.

The VMs run two types of applications. The first are interactive applications that continue running throughout the entire simulation and whose CPU and memory requirements over time are determined by equations (14) and (15) according to a previously established demand distribution [24].

$$CPUDemand_j(t) = (\frac{CPU_l}{2})(1 + u_m sin(\frac{2\pi t}{86400} - 2\pi s_m) \tag{14}$$

$$MemDemand_j(t) = \beta \times CPUDemand_j(t) \tag{15}$$

Applications of the second type are short-lived batch jobs whose CPU and memory usage are constant and equal to their maximum reserved capacity over their life time. Their lifetimes are determined by a truncated power law distribution with an exponent of 2, incremented by 10 minutes to account for VM boot time and capped at 24 hours (i.e. the length of a single simulation run). The rate at which batch requests arrive at the simulated data center is determined by a Poisson process with $\lambda = 0.016$.

We evaluated the system under different synthetic workloads with different Service-to-Batch (SB) ratios. The SB ratio defines the ratio of service applications to batch jobs. Workloads with SB ratios of 0.1, 0.4 and 0.6 were considered.

5.2 Logical Overlay

The PMs are logically connected by an overlay network, maintained by newscast. Each PM at each timestep only knows about $k = 20$ random neighbors, shaping its local view. We modified the newscast to provide a neighbor list, where the probability of having neighbors within the same server group is 0.6, compared to 0.3 for neighbors within the same cluster and 0.1 for random neighbors.

5.3 Control triggers

To avoid potential negative impacts of over-consolidation such as interferences and resource contentions, we aim for 75% resource utilization. To avoid stressing the system by triggering frequent overload triggers, we only trigger an overload event when a PM's CPU or memory utilization exceeds 90% for 10 consecutive cycles. We also assume that a PM is underloaded if its utilization in both dimensions drops below 30% for 10 consecutive cycles. To prioritize the control triggers, we set a 10 minute waiting time for underload triggers before they are processed. This allows the scheduler to delay the scheduling with the hope of placing newly arrived batch jobs to increase the load of underloaded servers instead of migrating VMs out from them.

5.4 Score function weight values

Unless stated otherwise, equal weight values of 0.2 were assigned to all five of the factors used to rank candidate VMs for consolidation (efficiency, risk, data transfer costs, imbalance and migration locality).

5.5 Performance parameters

The aim of the evaluation was to answer the following questions: How well are the applications packed? How does each of the score function variables (*risk*, *efficiency*, *migration locality*, *migration cost*, and *imbalance*) affect the stability and reliability of consolidation? And how effectively does the scheduler minimize the migration costs? To answer these questions, we measured the following performance parameters:

1. Average resource utilization:

$$U_{dc}(t) = \frac{\sum_{j=1}^{m} Res_j(t)}{n_{active} \times Capacity_{PM}}, \tag{16}$$

   where m is the number of placement requests at time t, $Res_j(t)$ is the CPU or memory usage of VM_j at time t, n_{active} is the total number of non-idle PMs, $Capacity_{PM}$ is the capacity of each PM.

2. Number of active servers at time t.

3. Average number of active servers:

$$average\ active\ servers(t) = \frac{\sum_{j=1}^{t} n_{active}(t)}{t} \tag{17}$$

4. Number of pending requests: The number of VM requests that are not scheduled within the assigned time frame.

5. Average imbalance rate: This parameter is a representation of resource stranding in the data center.

$$imb_{dc}(t) = \sum_{i=1}^{n} |CPU_i(t) - Memory_i(t)| \tag{18}$$

6. Total amount of data transfer (GB)

7. Batch job data transfer (GB): The volume of data transfered due to the cold migration of the batch jobs.

8. Service data transfer (GB): The volume of data transfered due to the live migration of the services.
9. Underload data transfer (GB): The volume of data transfered to free up the resources in an underloaded PM, either to open up space for larger jobs or to be put into a power saving mode.
10. Overload data transfer (GB): The volume of data transfered to resolve an overload event.
11. Total number of migrations
12. Number of batch job migrations
13. Number of service migrations
14. Number of undesirable state triggers: This parameter represents the durability and stability of the placement decisions and the number of underload or overload occurs as the result. The higher the number of triggers, the higher the need for reconfiguration and thus higher probability of increase in migrations.
15. Overload triggers: The number of triggered overload events.
16. Underload triggers: The number of triggered underload events.
17. Unresolved triggers: This parameter shows the number of underloads or overloads that has not been resolved.
18. Total migrations with distance=1: The number of migrations to a server within the same server group.
19. Total migrations with distance=2: The number of migrations to a server within the same cluster.
20. Total migrations with distance=3: The number of migrations to a server in another cluster.

6 Evaluation

We evaluated the proposed framework, which is henceforth referred to as RPM (Reconsolidating PlaceMent scheduler), through a series of experiments. We first compared the scheduling of resources by RPM to that achieved using a variation of the First-Fit Decreasing (FFD) [18, 25] bin packing heuristic,

Table 3: *A summary of the performance aspects and associated parameters measured to evaluate the packing of the applications and the scheduler's performance*

Performance aspect	**Performance parameter**
Consolidation efficiency	Average resource utilization Average number of active servers Average imbalance rate Number of pending requests
Durability of consolidation	Number of undesirable state triggers Overload state triggers Underload state triggers Unresolved triggers
Data transfer efficiency	Total amount of data transfer Batch job data transfer Service data transfer Underload data transfer Overload data transfer
Migration count	Total number of migrations Number of batch migrations Number of service migrations
Migration locality	Total migrations within the same group (dist=1) Total migrations within the same cluster (dist=2) Total migrations to the other cluster (dist=3)

known as *FFD-sum*. We also investigated the impact of each of the variables used in the heuristic function of RPM on the performance parameters listed in Table 3.

6.1 Comparison with multi-dimensional FFD

In the first experiment, we compared the performance of RPM to that of *FFD-sum*, an extension of *FFD*, one of the most common VM consolidation heuristics. The comparison was made with respect to the criteria presented in Table 3. FFD orders the PMs and the VM requests in a decreasing order of size and places each VM on the first PM with sufficient capacity in the list. Ordering jobs by their CPU or memory requirements causes resource stranding, and the scheduler favors packing on one dimension over the other [8]. To support multiple resource dimensions, we used an extension of FFD known as *FFD-sum* [18].

As shown in Equation 19, *FFD-sum* maps the vector of capacities into a single scalar, using a weighted sum of the values on each dimension of CPU and memory:

$$Volume(V) = \sum\nolimits_{r_i \in Resources} w_i \times r_i, \tag{19}$$

where w_i is the weight assigned to each resource and reflects the scarcity of the resource. The weight is defined as the ratio of the total demand r_i to the available capacity h_i,

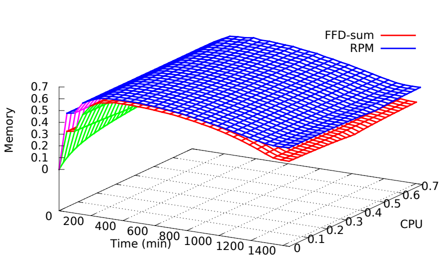
$$w_i = \sum r_i / h_i. \tag{20}$$

Figure 3 and Table 4 compare the performance of FFD-sum and RPM with respect to selected metrics. Figure 3a shows the average CPU and memory utilization of the data center over time for each scheduler. The RPM scheduler improved the average CPU and memory utilization by reducing the number of active PMs required to serve the workload. However, its main beneficial effect was that it reduced the cost of maintaining a high packing efficiency. As shown in Table 4, RPM reduced the number of sub-optimal state triggers by up to 60% by accounting for the risks of variability and resource contention. Figure 3b shows how the number of migrations varied over time. The durable scheduling decisions made by RPM reduced the number of migrations required to resolve the sub-optimal states.

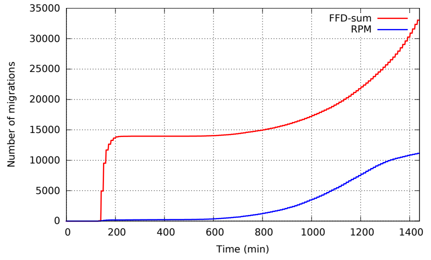
Moreover, RPM accounts for the difference in latency sensitivity between batch jobs and interactive services by selecting a suitable migration policy (i.e. cold or live migration) for jobs of each type. While live migration is necessary for interactive services to guarantee their responsiveness during migration, it imposes additional costs caused by re-transfer of dirtied pages. This assumes that live migration is performed using the pre-copy approach implemented in contemporary hypervisors. If post-copy live migration is used instead, other considerations should be taken into account [26]. The latency sensitivity of interactive services and their extra data transfer costs makes them less appealing options for migration. RPM accounts for this by calculating the amount of data to be transferred for a given job based on the migration policy for jobs of the relevant class and the transfers of dirty pages required for live migrations.

Figure 3c shows the volume of transferred data associated with services and batch jobs under RPM and FFD-sum during the simulation time. It is clear that choosing a migration strategy based on the application's latency sensitivity and carefully estimating the associated migration costs reduced the number of service migrations and the volume of service data transferred over the course of the simulation.

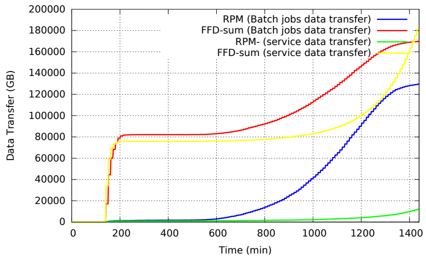
The bulk of the data transfer observed when using the RPM scheduler stems from the transfer of batch jobs. The volume of data transferred in such cases could be reduced further by adopting a stop-and-restart policy for batch jobs



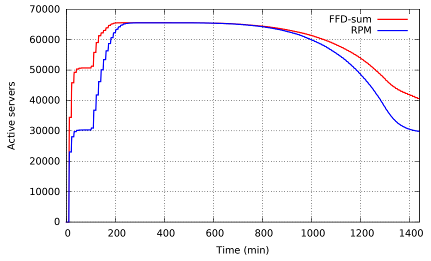
(a) Resource utilization under FFD-sum and RPM



(b) Number of migrations under FFD-sum and RPM



(c) Batch and service data transfer under FFD-sum and RPM



(d) Number of active PMs under FFD-sum and RPM

Figure 3: Comparison of FFD-sum and RPM

whereby they would be rescheduled onto another machine without actually performing a migration. In contrast, the disregard of job type by FFD-sum requires the transfer of more latency- and SLA-sensitive interactive services than is the case under RPM.

It is also clear that RPM was more successful than FFD-sum at resolving sub-optimal states. This is mainly because RPM accounts for the proportionality of resource usage when making placement decisions and thus avoids resource stranding and fragmentation. Finally, the number of overload triggers generated under RPM was greater than under FFD-sum due to the higher utilization achieved under RPM. However, all of the overload triggers were resolved and no actual overload occurred. RPM also reported lower numbers of pending requests than FFD-sum. We define pending requests as VM placement requests that could not be scheduled within the defined computation time.

We also evaluated the system's performance in cases featuring different ratios of batch jobs to services. The greater the proportion of batch jobs, the greater the volatility of the workload and thus the greater the need for frequent schedul-

Table 4: *A performance comparison of the FFD-sum and RPM schedulers*

Parameter	FFD-sum	SE	RPM	SE
# of active servers	40504	22.94	29859.6	12.0
Total amount of data transfer	351962.2	1138.69	141554	902.03
Total # of migrations	33187	48.30	11092	63.24
Total # of batch migrations	11756	53.22	9797	61.49
Total # of service migrations	21431	31.55	1295	14.18
Total # of rejected requests	10657	24.21	6254.8	48.85
Batch data transfer (GB)	169781.24	961.39	129834.11	886.24
Service data transfer (GB)	182186.15	493.89	11722.67	157.21
Sub-optimal state triggers	26574	36.21	10873.8	48.84
Overload triggers	29	1.35	73.8	4.30
Unresolved triggers	278	9.09	18.8	1.56

ing. This is because batch jobs usually have short lifetimes and impose a higher job turn-over due to their frequent arrivals and terminations. Consequently, more batch jobs means greater probabilities of resource fragmentation and sub-optimal resource distribution. A sub-optimal resource distribution forces the scheduler to respond by moving VMs more frequently. As seen in Table 5 the amount of *data transfer per trigger* for different SB ratios is comparable in all scenarios.

Table 5: *Data transfer per trigger for different workloads with different SB-ratios*

SB ratio	Sub-optimal state triggers	Data-Transfer per trigger
0.1	20152	12.42 GB
0.4	10873	12.73 GB
0.6	7051	12.98 GB

6.2 Impact of score function variables

To understand the impact of each of the five variables of the score function used for placement decisions, we investigated each variable individually by adjusting the weighting applied to them when computing the score function. Weights in the range $\lambda = \{0.0\ ,\ 0.8\}$ were investigated for each variable in Equation 3.

6.2.1 Efficiency

Table 6 shows the performance metrics impacted by changing the weighting of the *efficiency* factor in Equation 3. Considering efficiency of placement while selecting servers increased the average utilization and reduced the active number of servers required to serve a specific load. Interestingly, the main impact of introducing efficiency was not due to improvements in utilization but to a reduction in the number of underload triggers, which reduced the volume of data transfer required to resolve the underload state. Efficient packing also reduced the number of unresolved triggers by up to 70%. However, this greater consolidation came at the cost of increasing the number of overload triggers.

Table 6: *Impact of the efficiency factor on packing efficiency*

Efficiency weight	**0.0**	**SE**	**0.8**	**SE**
Average CPU	68%		71%	
Active servers	57659	7.67	54263	19.80
Underloads triggers	50159	151.66	10941	32.61
Unresolved triggers	158.75	2.28	35	2.60
Overload triggers	50	3.01	147	6.89
Underload data-transfers	480948	900.14	135081	426.44
Data Transfer	481364	893.83	136248	433.95

6.2.2 Risk

We evaluated the impact of considering the risk of load variability and resource contention on possible future overloads, migrations and data transfer. Table 7 shows the numbers of overload triggers and overload data transfers as well as the average number of active servers during the simulation runtime for different risk weightings. Accounting for risk in the score function reduced the number of overload triggers by up to 35% and the volume of data transfer due to offload by up to 33%. However, this came at the cost of a 2% increase in the average number of active servers.

Table 7: *Impact of the risk factor on packing efficiency*

Risk weight	**0.0**	**SE**	**0.8**	**SE**
Overload triggers	81	4.6	52	3.8
Overload data-transfers	652.0	30.51	434.20	32.09
Active servers	54099	14.9	55063	5.7

6.3 Imbalance

We examined the consequences of resource stranding and the impact of accounting for the imbalance factor as a way of mitigating the negative effects of stranding. As shown in Table 8, taking the imbalance factor into account improved resource utilization and reduced the number of pending requests. Moreover, placing complementary workloads together reduced the probability of overload and thus the number of overload triggers.

Table 8: *Impact of the imbalance factor on packing efficiency*

Imbalance weight	**0.0**	**SE**	**0.8**	**SE**
Average CPU	68%	▨	70%	▨
Average memory	68%	▨	70%	▨
Pending requests	10115.8	45.75	6208.6	36.83
Average imbalance	8.15	0.01	6.17	0.01
Dist2	496	10.58	2140	26.86
Overload triggers	151	9.57	69	4.61
Underload triggers	13408.6	11.61	11637	44.37
Data Transfer	173561	321.90	140486	709.02

6.3.1 Migration cost

To evaluate the impact of the migration cost factor, we measured the total data transfer volume and the data transfer volume caused by offloading the overloaded PMs using different migration cost weightings. Interestingly, accounting for individual migration costs in this way during the decision-making process had no appreciable impact on the overall migration cost. Within our experimental setting, the total volume of data transferred declined by only 1% when weighting migration cost heavily. Moreover, this small improvement was mainly due to a reduction in the volume of data transferred during offloads, which was achieved by distinguishing between VMs of different sizes, types and dirty rates.

The low impact of the *migration cost* variable is due to the fact that the overall migration cost is mainly dependent on the number of migrations that are performed rather than the volume of data transferred during a single migration. Moreover, the need to select a VM for migration only occurs during overload situations, which are much less frequent than underload situations. During an underload situation, the scheduler's objective is to evacuate the underloaded PM by migrating every VM running on it, irrespective of type and size. Because there is no VM selection process in such situations, the migration cost factor has no opportunity to have any effect.

An interesting observation follows from comparing the impacts of the efficiency, imbalance, and migration cost factors on the data transfer volume are shown in Tables 6, 8, and 9 respectively. Somewhat counter-intuitively, we observe that the most significant migration cost reductions are achieved by

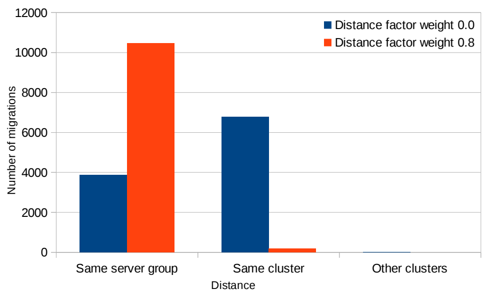


Figure 4: Numbers of migrations performed within groups, between groups, and between clusters with different distance factor weightings

preventing the occurrence of undesirable states rather than careful VM selection once a migration becomes necessary. Undesirable states are prevented from developing by striving for an efficient initial deployment and avoiding resource stranding rather than careful VM selection, when a migration is needed. This finding confirms the advantages of careful semi-static consolidation [27] relative to dynamic consolidation, or in other words, the advantage of prevention over treatment.

Table 9: *Impact of the migration cost factor on packing efficiency*

Migration cost weight	**0.0**	**SE**	**0.8**	**SE**
Data transfer	142356.6	425.21	139718.2	663.13
Overload data transfer	748.36	46.2	389.13	52.2

6.3.2 Migration Locality

We compared the number of migrations performed within groups, (dist = 1), between groups, (dist = 2) and between clusters, (dist = 3) under different weightings of the migration locality factor.

The results shown in Figure 4 and Table 10 demonstrate increasing the weighting of the migration locality factor increases the proportion of local migrations within a server group relative to those of migrations between groups or clusters. However, it slightly increases the migration cost in terms of data transfer and also the number of sub-optimal state triggers due to the associated trade-off with risk and migration cost.

Table 10: *Impact of the locality factor on packing efficiency*

Locality weight	**0.0**	**SE**	**0.8**	**SE**
Dist1	3879	8.28	10470.6	45.10
Dist2	6779.25	30.10	195.6	3.45
Dist3	10.25	0.8	0	0
Data Transfer (GB)	127698.7	337.60	156863.2	778
Undesirable state triggers	9944	21.91	12267.4	54.37

7 Related work

Several previous studies have explored different aspects of the consolidation problem, providing different formulations based on centralized and distributed approaches, emphasizing different objectives, and factoring different aspects of consolidation into their models. We classify these works primarily in terms of their centralization or decentralization. However, we also briefly discuss the formulation, objectives, and migration awareness of each approach.

7.1 Centralized approaches

The first group of centralized approaches are those that use linear programming models to solve the scheduling problem. These models produce exact solutions but the polynomial-time complexity of the solution hinders their applicability to real size problems. Speitkamp and Bichler [5] formulated consolidation as an optimization problem in an attempt to minimize server costs while respecting capacity constraints. They approached consolidation as a one-time decision-making process, ignoring practical issues such as variable load demands and migration costs including network contention due to migrations.

Ghribi et al. [4] also used an integer linear programming model to produce two exact algorithms for consolidation. Their objective was to achieve energy efficiency in cloud data centers while limiting the number of migrations. They used semantic rules such as “avoid migrations of VMs that have almost finished running their jobs” or “avoid redundant VM migrations” in order to limit the number of migrations. The convergence time of the algorithms used in this approach grows exponentially as the number of VM requests or number of servers increases, making it impractical for large data centers. In addition, these authors modeled the migration cost as the cost of power consumption imposed during migration. Thus, while the cost of migration is accounted for after a fashion, the cost model is blind to the application type, amount of data transferred, and possible bandwidth contentions.

Similarly, Ferreto et al. [28] provided a LP formulation for consolidation with the aim of controlling VM migration. Their approach prioritizes VMs with steady capacity over those with variable capacity, and avoids migrations of steady capacity VMs. Aside from the high computational complexity of this approach, we feel that it is sub-optimal because accounting for load variability is necessary but not sufficient when planning VM migrations.

The second group of studies examined different heuristics for VM consolidation. Beloglazov et al, [29] modified the Best Fit Decreasing (MBFD) heuristic to achieve energy efficiency in cloud data centers. Their algorithm schedules resources based only on their CPU utilization (disregarding memory utilization) and thus makes the scheduling problem one-dimensional. They also tried to minimize the number of VM migrations by selecting the largest VMs to resolve overload situations. The proposed VM selection algorithm is application-agnostic and does not account for the difference in latency sensitivity between VMs running service and batch jobs.

Mistral [30] is a holistic optimization framework for VM consolidation that is designed to strike a balance between power usage and application performance. It can implement a range of adaptive measures such as vertical scaling of a VM's CPU as well as adding or removing VMs, shutting them down or restarting them, and performing live migrations. The adaptation cost is defined in terms of power and performance overheads, which are quantified by measuring changes in response times and power usage when two workloads are co-located across different physical hosts. The A* algorithm is used to search the configuration space and find a sequence of adaptation measures that transform the current configuration into the newly identified optimal configuration. Unfortunately, this framework does not scale well with increasing numbers of VMs and servers when the configuration space for VM placement is very large.

Svärd et al. [31] also discussed a set of heuristics designed to maintain the optimality of allocations using a similar set of actions to those available under Mistral. These actions are triggered when a PM crashes or a VM arrives or exits. This approach has a centralized design and supports at most 48 PMs. Its heuristics are not topology- or risk-aware.

Sandpiper [32] offers black box and grey box approaches for hotspot detection, the identification of new mappings between VMs and resources, and the initiation of necessary migrations. It profiles VMs and servers using OS- and application-level statistics, and defines the migration overhead as the amount of data transferred. While it represents an interesting model, Sandpiper's main purpose is to resolve hotspots rather than to achieve consolidation per se.

pMapper [33] introduced an application placement controller to minimize power consumption in a heterogeneous virtualized server cluster. It defines migration cost as the decrease in throughput due to live migration and the estimated revenue loss due to the associated decline in performance. The heuristic used in pMapper is FFD and resources are modeled as one-dimensional compute resources. In contrast to our approach, the heuristic is neither topology- nor risk-aware.

Sheikhalishahi et al [34] introduced a heuristic function to schedule multidimensional resources. Their main aim is to increase resource efficiency and to reduce the waiting and slowdown time by improving resource stranding and re-ordering the jobs in the queue. The proposed heuristic is not topology and migration aware.

Finally, in the course of our work, we found it useful to study previous attempts to develop optimized migration policies as well as reports focused on consolidation. Jain et al [35] address the problem of topology-aware VM migration in a multi-rooted tree data center, and use VM migration to minimize the the number of hot servers in the data center. They formulated the problem as one of constrained migration and tried to compute the maximal set of hot servers that can be relieved by migrating a subset of their VMs. They showed that it is possible to alleviate the server hotspots via short migration paths. Remedy [15] is a network aware VM management system whose primary purpose is to minimize the cost of migration. A detailed migration cost model is discussed that takes a range of factors into account, including the VM's image size, page dirty rate, available bandwidth, and migration completion deadline. It should again be noted that the primary focus of these two works was on migration efficiency rather than packing efficiency.

7.2 Distributed approaches

The other group of studies focused on addressing the scalability problem while still meeting performance goals. This is generally achieved by adopting a formulation in which the task is treated as a distributed resource management problem where each manager solves the problem with respect to its incomplete view of the system.

Sparrow [36] is a stateless decentralized scheduler whose purpose is to improve the throughput and availability of highly parallel jobs with low latency. It uses random sampling to discover available resources within a cluster. Although its conceptual foundations are somewhat similar to those of RPM, Sparrow's main objective is to maximize availability rather than resource utilization and consolidation. In addition, it is oriented towards short tasks and therefore does not address the complexities of migration because it assumes that all jobs can simply be re-scheduled. In contrast, RPM is also capable of handling long-running services with variable demand. Moreover, Sparrow is designed to operate within a statically partitioned cluster and thus does not deal with the fragmentation problem on the cluster level, whereas RPM uses topology-aware sampling to reduce the negative impacts of static partitioning and cluster-level fragmentation.

Wuhib [37] proposed a resource management architecture for optimal resource allocation. The architecture is composed of a set of cooperative controllers interacting via a gossip protocol. The P2P nature of the architecture is similar to our approach. However, they did not consider migration costs, topological constraints, or the risk of variability in their model

Snooze [38] introduced a self-organizing static hierarchical architecture for distributed VM management. The static hierarchical design of the architecture limits the packing efficiency because it lacks inter-group coordination. The authors extended their work to overcome the problem of static partitioning by adopting an unstructured P2P overlay. However, they did not enforce any topological constraints on their overlay so large numbers of costly inter-cluster migrations may occur and the probability of network contention on oversubscribed links can increase substantially.

V-Man [39] is a fully decentralized algorithm for VM consolidation that uses a gossip protocol to achieve efficiency and scalability. V-Man consolidates VMs solely on the basis of their CPU utilization and thus favors packing along one dimension, leading to resource stranding. Like the model proposed by Wuhib et al. [37], V-Man does not account for migration costs, topology constraints, or the risk of resource contention.

8 Conclusion

The paper presents and evaluates a decentralized resource management framework for large-scale cloud infrastructures. The P2P structure of the framework provides parallelization, a high degree of concurrency and provides reasonable scalability as the number of PMs and VMs increases. Moreover, the P2P architecture's random overlay allows the system to create a logical dynamic connectivity among a large pool of resources and reduce the negative impacts of static partitioning, which lead to low utilization. It also accounts for the physical proximity of the neighbors when building the logical overlay, thereby reducing the costs of network transit and reconfiguration.

As part of the framework, we have introduced a scheduling heuristic that takes into account relevant decision factors to produce efficient and durable packing. The heuristic function estimates the efficiency of packing for different candidates, factoring in their physical proximity based on the data center's topology as well as their risks of resource contention and load variability.

It also estimates the migration costs for individual VMs based on the type of application running within the VM, its dirtying rate, and the preferred migration strategies for jobs of specific types. Using these estimates, the heuristic function proposes a candidate destination PM, generates a migration plan, and determines the necessary migration actions, the destination PM, and the set of VMs to be migrated.

We have evaluated the framework by simulating a data center using PeerSim [21]. The data center is structured as a three-level multi-rooted tree comprised of 65536 PMs serving 230000 VMs over a simulation time of 24 hours and facing a changing load caused by VM churn and changes in the VMs' demand. We compared our risk-aware, topology aware heuristic (RPM) to the state of the art FFD-sum algorithm. RPM achieved a slightly higher resource utilization than FFD-sum, but its main advantage was its ability to reduce the cost of maintaining the desired packing efficiency. By accounting for the risks of variability and load contention, RPM reduces the number of sub-optimal state triggers by up to 60% compared to FFD-sum and thus reduces the number of migrations required to resolve the sub-optimal state. RPM reduces migration costs by distinguishing between different job types (batch jobs and interactive services) on the basis of their latency sensitivity, and by selecting a suitable migration policy (cold vs. live migration) for jobs of different types. It also reduces the number of service migrations and the volume of service data transfers. In addition, we showed that RPM reduces the negative impacts of resource stranding by accounting for the proportionality of resource usage when selecting a PM candidate.

Finally, we studied the impact of the individual variables considered within RPM's heuristic function, identifying key performance trade-offs. For example, increasing the weighting of the efficiency factor improves the utilization but also increases the probability of triggering overloads. Similarly, increasing the weighting of the risk factor reduces the number of overload triggers at the cost of increasing the number of active PMs. However, increasing the weighting of the migration cost variable when selecting a VM set for migration had no appreciable impact on the ultimate migration cost. Our results show that greatest reductions in migration cost are achieved by preventing undesirable states from developing in the first place rather than from carefully selecting the optimal VMs to migrate once the system has entered an undesired state. This can be accomplished by striving for an efficient initial deployment and avoiding resource stranding. This demonstrates the benefits of careful semi-static consolidation over dynamic consolidation or, in other words, the advantage of prevention over treatment.

Acknowledgments

Financial support for this work was provided in part by the Swedish Research Council (VR) under contract number C0590801 for the project Cloud Control, the Swedish Government's strategic research project eSSENCE, and the European Union's Seventh Framework Programme under grant agreement 610711 (CACTOS).

References

[1] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: flexible, scalable schedulers for large compute clusters," in *Proceedings of the 8th ACM European Conference on Computer Systems*, pp. 351–364, ACM, 2013.

[2] J. L. Berral García, R. Gavaldà Mestre, J. Torres Viñals, *et al.*, "An integer linear programming representation for data-center power-aware management," 2010.

[3] W. Li, J. Tordsson, and E. Elmroth, "Virtual machine placement for predictable and time-constrained peak loads," in *Economics of Grids, Clouds, Systems, and Services*, pp. 120–134, Springer, 2012.

[4] C. Ghribi, M. Hadji, and D. Zeghlache, "Energy efficient VM scheduling for cloud data centers: exact allocation and migration algorithms," in *13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pp. 671–678, IEEE, 2013.

[5] B. Speitkamp and M. Bichler, "A mathematical programming approach for server consolidation problems in virtualized data centers," *IEEE Transactions on Services Computing*, vol. 3, no. 4, pp. 266–278, 2010.

[6] M. Sedaghat, F. Hernández-Rodriguez, and E. Elmroth, "Autonomic resource allocation for cloud data centers: a peer to peer approach," in *The ACM Cloud and Autonomic Computing Conference (CAC'14)*, pp. 131–140, ACM, 2014.

[7] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, 2013.

[8] A. Verma, M. Korupolu, and J. Wilkes, "Evaluating job packing in warehouse-scale computing," in *IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 48–56, IEEE, 2014.

[9] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, “Live migration of virtual machines,” in *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pp. 273–286, USENIX Association, 2005.

[10] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel, “The cost of a cloud: research problems in data center networks,” *ACM SIGCOMM computer communication review*, vol. 39, no. 1, pp. 68–73, 2008.

[11] M. Jelasity and M. Van Steen, “Large-scale newscast computing on the internet,” tech. rep., Citeseer, 2002.

[12] A. W. Richa, M. Mitzenmacher, and R. Sitaraman, “The power of two random choices: A survey of techniques and results,” *Combinatorial Optimization*, vol. 9, pp. 255–304, 2001.

[13] L. Tomás and J. Tordsson, “Cloudy with a chance of load spikes: Admission control with fuzzy risk assessments,” in *IEEE/ACM 6th International Conference on Utility and Cloud Computing (UCC)*, pp. 155–162, IEEE, 2013.

[14] B. Viswanathan, A. Verma, and S. Dutta, “Cloudmap: workload-aware placement in private heterogeneous clouds,” in *Network Operations and Management Symposium (NOMS),*, pp. 9–16, IEEE, 2012.

[15] V. Mann, A. Gupta, P. Dutta, A. Vishnoi, P. Bhattacharya, R. Poddar, and A. Iyer, “Remedy: Network-aware steady state VM management for data centers,” in *Networking*, pp. 190–204, Springer, 2012.

[16] H. Chen, H. Kang, G. Jiang, and Y. Zhang, “Network-aware coordination of virtual machine migrations in enterprise data centers and clouds,” in *IFIP/IEEE International Symposium on Integrated Network Management (IM 2013)*, pp. 888–891, IEEE, 2013.

[17] S. Setty, “vMotion architecture, performance, and best practices in VMware vsphere 5,” *VMware, Inc., Tech. Rep*, 2011.

[18] S. Lee, R. Panigrahy, V. Prabhakaran, V. Ramasubramanian, K. Talwar, L. Uyeda, and U. Wieder, “Validating heuristics for virtual machines consolidation,” *Microsoft Research, MSRTR-2011-9*, 2011.

[19] S. He, L. Guo, M. Ghanem, and Y. Guo, “Improving resource utilisation in the cloud environment using multivariate probabilistic models,” in *5th IEEE International Conference on Cloud Computing (CLOUD)*, pp. 574–581, IEEE, 2012.

[20] M. Sedaghat, F. Hernández-Rodriguez, E. Elmroth, and S. Girdzijauskas, "Divide the task, multiply the outcome: Cooperative VM consolidation," in *In Proceedings of The 6th IEEE International Conference on Cloud Computing Technology and Science (CloudCom 2014)*, pp. 300–305, IEEE, 2014.

[21] A. Montresor and M. Jelasity, "Peersim: A scalable p2p simulator," in *Ninth IEEE International Conference on Peer-to-Peer Computing, 2009. P2P'09.*, pp. 99–100, IEEE, 2009.

[22] "EC2 instance types." `http://aws.amazon.com/ec2/instance-types/`.

[23] "EC2 instance bandwidths." `http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ebs-ec2-config.html`.

[24] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Heterogeneity and dynamicity of clouds at scale: Google trace analysis," in *Proceedings of the Third ACM Symposium on Cloud Computing*, p. 7, ACM, 2012.

[25] B. S. Baker, "A new proof for the first-fit decreasing bin-packing algorithm," *Journal of Algorithms*, vol. 6, no. 1, pp. 49–70, 1985.

[26] P. Svärd, B. Hudzia, S. Walsh, J. Tordsson, and E. Elmroth, "Principles and performance characteristics of algorithms for live VM migration," *ACM SIGOPS Operating Systems Review*, vol. 49, no. 1, pp. 142–155, 2015.

[27] A. Verma, J. Bagrodia, and V. Jaiswal, "Virtual machine consolidation in the wild," in *Proceedings of the 15th International Middleware Conference*, pp. 313–324, ACM, 2014.

[28] T. C. Ferreto, M. A. Netto, R. N. Calheiros, and C. A. De Rose, "Server consolidation with migration control for virtualized data centers," *Future Generation Computer Systems*, vol. 27, no. 8, pp. 1027–1034, 2011.

[29] A. Beloglazov, J. Abawajy, and R. Buyya, "Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing," *Future Generation Computer Systems*, vol. 28, no. 5, pp. 755–768, 2012.

[30] G. Jung, M. A. Hiltunen, K. R. Joshi, R. D. Schlichting, and C. Pu, "Mistral: Dynamically managing power, performance, and adaptation cost

in cloud infrastructures," in *30th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pp. 62–73, IEEE, 2010.

[31] P. Svärd, W. Li, E. Wadbro, J. Tordsson, and E. Elmroth, "Continuous datacenter consolidation," 2014.

[32] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif, "Sandpiper: Black-box and gray-box resource management for virtual machines," *Computer Networks*, vol. 53, no. 17, pp. 2923–2938, 2009.

[33] A. Verma, P. Ahuja, and A. Neogi, "Pmapper: power and migration cost aware application placement in virtualized systems," in *Middleware 2008*, pp. 243–264, Springer, 2008.

[34] M. Sheikhalishahi, R. M. Wallace, L. Grandinetti, J. L. Vazquez-Poletti, and F. Guerriero, "A multi-dimensional job scheduling," *Future Generation Computer Systems*, 2015.

[35] N. Jain, I. Menache, J. S. Naor, and F. B. Shepherd, "Topology-aware VM migration in bandwidth oversubscribed datacenter networks," in *Automata, Languages, and Programming*, pp. 586–597, Springer, 2012.

[36] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, "Sparrow: distributed, low latency scheduling," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pp. 69–84, ACM, 2013.

[37] F. Wuhib, R. Yanggratoke, and R. Stadler, "Allocating compute and network resources under management objectives in large-scale clouds," *Journal of Network and Systems Management*, vol. 23, no. 1, pp. 111–136, 2015.

[38] E. Feller, L. Rilling, and C. Morin, "Snooze: A scalable and autonomic virtual machine management framework for private clouds," in *Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, pp. 482–489, IEEE Computer Society, 2012.

[39] M. Marzolla, O. Babaoglu, and F. Panzieri, "Server consolidation in clouds through gossiping," in *IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, pp. 1–6, IEEE, 2011.

Paper V

A Virtual Machine Re-packing Approach to the Horizontal vs. Vertical Elasticity Trade-off for Cloud Autoscaling

M. Sedaghat, F. Hernandez-Rodriguez, and E. Elmroth

In Proceedings of the ACM Cloud and Autonomic Computing Conference, (CAC'13),
Article no. 6, ACM, 2013

A Virtual Machine Re-packing Approach to the Horizontal vs. Vertical Elasticity Trade-off for Cloud Autoscaling[*]

Mina Sedaghat[†], Francisco Hernandez-Rodriguez[†], Erik Elmroth[†]

Abstract

An automated solution to horizontal vs. vertical elasticity problem is central to make cloud autoscalers truly autonomous. Today's cloud autoscalers are typically varying the capacity allocated by increasing and decreasing the number of virtual machines (VMs) of a predefined size (horizontal elasticity), not taking into account that as load varies it may be advantageous not only to vary the number but also the size of VMs (vertical elasticity). We analyze the price/performance effects achieved by different strategies for selecting VM-sizes for handling increasing load and we propose a cost-benefit based approach to determine when to (partly) replace a current set of VMs with a different set. We evaluate our repacking approach in combination with different auto-scaling strategies. Our results show a range of 7% up to 60% cost saving in total resource utilization cost of our sample applications and workloads.

Keywords: Cloud computing; Autoscaling; Autonomous computing; Vertical elasticity; Horizontal elasticity.

1 Introduction

Autonomous management systems are key to the realization of future elastic cloud infrastructures, both due to the scale and complexity of the infrastructures and due to the fact that management actions often may need to be performed with only seconds or minutes notice. Key to such an autonomous management

[*]The paper has been re-typeset to match the thesis style.

[†]Dept. of Computing Science, Umeå University, SE-901 87 Umeå, Sweden, email: {mina, francisco, elmroth}@cs.umu.se

system is the ability to automatically scale the amount of resources allocated to an application, depending on its load, in order to appropriately handle a hosted application's load peaks without over-provisioning when load is low. The scaling can be done either by changing the number of VMs, *horizontal elasticity*, or by changing the size of VMs, *vertical elasticity*. Recently, most attention has been given to horizontal elasticity management, partly due to the fact that vertical elasticity is much more limited as it cannot scale outside single physical machines.

However, as the horizontal elasticity decisions also have to consider what size VMs to initiate or terminate, the problems are not fully detached. For a certain amount of capacity there is some set of VMs (of possibly varying but normally predefined sizes) that most cost-efficiently provides that capacity. Even if during a sequence of scale-up operations the autoscaler determines the most cost-efficient way to provide the extra capacity needed at each step, the sequence of such operations may result in a far from optimal VM set for the aggregate capacity. Hence, at some point in time the set of VMs allocated would benefit from being replaced by a new, more optimized set of VMs providing the same capacity, here called *repacking* of VMs. Alternatively, each autoscaling decision could take into account not only adding VMs but also replacing some VMs with VMs of other sizes. As frequent turning on and off VMs is costly, this latter approach is not suitable in scenarios where frequent scaling operations are performed.

In this contribution, we as part of a holistic cloud management system [1], investigate the problem of when and how a set of allocated VMs should be repacked to a new optimal set of VMs and propose a model to provide the required capacity more cost-efficiently. The solution takes into account the price and capacity of a number of predefined VMs, where capacity most commonly is based on the specific application's performance on a VM of that type. Of course, price-changes may occur over time and that by itself is a reason for re-allocation. Moreover, we analyze the different parameters that affect the repacking decisions, including *price/performance* ratio of an application, cost of reconfigurations, behavior of different autoscalers, and the relation of repacking with expected life-span of the reconfigured VM set before scaling down or before the next reconfiguration.

2 Background and motivation

Applications with capacity demands with significant variations over time benefit the most when deployed in cloud environments, since they can benefit from the elasticity provided by clouds. In this contribution we assume that elasticity is handled through autoscaling.

Public cloud providers offer a variety of instance types, each characterized by different sizes and different prices. For example, Amazon EC2 [2] currently offers 17 instance types that range from standard instances to application specific instances offering high I/O, high CPU, or high memory. For private clouds we also assume that VMs of different sizes, prices, and price/performance ratios can be provisioned.

Different autoscaling mechanisms use different techniques to estimate and predict the capacity required to serve an application with a changing demand [3–7]. However, current autoscalers decide on when and how to change the capacity for a changing demand, but normally not evaluate which instance-type to choose. For example, autoscalers like those offered by Amazon or Rightscale [8] only consider the number of VMs to provision. In these cases, the user has to specify what VM-type to add or remove when a condition given by a threshold is met [6]. Thresholds and rules that act upon them must be defined by a user with a good understanding of both workload and application. With more information available, more advanced autoscalers can make improved decisions such as finding a cost efficient resource set [9, 10]. However, even for advanced autoscalers, when demand changes prior optimal combination of VMs becomes sub-optimal.

Notably, the cost-efficiency for different VM-types depends on pricing and application performance. In case there are preferences, e.g., on keeping the total number of VMs low, we can assume, without loss of generality, that this is included in an application's price/performance ratio. In case the best price/performance ratio is provided by the smallest VM-type, taking into account that this also leads to the large number of VMs, the VM-type selection problem can in practice be avoided by always selecting the smallest VM type and never make any re-configurations. Of more interest, from a re-configuration point of view, is when the price/performance ratio is more advantageous for large VM-types. In particular, this effect can be large when there is a penalty associated with having large number of VMs. For such scenarios, there is an inherent conflict in avoiding costly over-provisioning by allocating too much capacity and by allocating more costly small VMs.

In this contribution we consider different strategies for balancing this trade-off through selection of VM-type to add when more capacity is needed and with reconfiguration of the total VM-set as it has become less cost-efficient after a series of small capacity increases. It should be remarked that this contribution makes no assumptions on whether the autoscaler used is reactive [2, 8] or proactive [5,9]. It only considers how to most cost-efficiently provide the requested capacity, regardless of how that request has been determined.

3 Repacking approach

We formulate the problem of determining how to reconfigure the VM set as follows:
There is always a cost per time-unit associated with resource usage, regardless if the hosting is done on external resources and the cost is simply the explicit given price or if the price is a combination of investment and running cost on local machines in a private cloud. For a given VM-type, there is also an associated capacity, which may, preferably, be expressed in application specific terms. We assume, without loss of generality, that there are n VM-types VM$_i$, $i = 1, \ldots, n$ with an associated price p_i per time unit, during which they are able to perform work represented by the capacity c_i and the application requires at least the capacity C_{Req}. Assume that there is currently a set S of n VMs that provides the aggregated capacity $C \geq C_{Req}$ at the total price $P = \sum_{i=1}^{n} p_i$. We are interested to find a new set $\hat{S}$ with $\hat{n}$ VMs, that provides a capacity $\hat{C} \geq C_{Req}$ at a minimum price $\hat{P}$, and $\hat{P} < P$, while considering an overhead cost of reconfiguration R. The reconfiguration cost is to compensate the impact of reconfiguration on the application performance and may, if so required, include other costs than only the pure resource cost. Finally, given an expected stability duration d before the capacity requires another change, we want to determine if $\hat{P}$ is sufficiently much smaller than P to motivate the cost for the reconfiguration. Hence, the problem in its most basic form is:

$$1 - [(\hat{P} \times d + R)/(P \times d)] > K \tag{1}$$

where K is gain factor that defines the minimum benefit that a human manager defines as the threshold to perform reconfiguration. The intention of considering K is to avoid unnecessary costly changes for a marginal gain.

In this section we introduce a model for determining when to perform *Repacking* (or re-configuration) of the VM-set used to provide a certain capacity. Repacking of an application with a changing workload is a process that should be performed repeatedly and it is not a one time decision, so the following steps are executed repeatedly.

3.1 Finding the optimal Set

The goal of repacking is to find a configuration of resources that minimizes the resource utilization cost, given current configuration, a number of different VM-types with different price/performance characteristics, a cost-function for performing repacking, and an expected time (duration) for which a new configuration is expected to be used.

The first step is to find an optimal configuration defined in terms of number of VMs of each type that are required to serve the application with the desired QoS and with the minimum cost. In other words, we need to find how many instances are required and of what type they should be to serve the load with a minimal resource provisioning cost. The problem is formulated as a combinatorial optimization problem:

$$Minimize \;\; Price = \sum_{i=1}^{n} n_i \times p_i$$

Subject to:

$$\sum_{i=1}^{n} n_i \times c_i \geq C_{Req}$$

$$n_i \in N$$

where p_i and c_i are respectively the price and the capacity of VM type i; n is the number of VM types; C_{Req} is the capacity required to provide the desired performance, and n_i is the number allocated instances of VM type i. The capacity of a VM, c_i, is presented as the average capacity that each VM can serve for the particular hosted application and it is calculated as the amount of work that saturates the instance. We assume that each instance type can serve a portion of load defined in terms of number of requests and proportional to its capacity (i.e., memory and CPU capacity). In case there is a cost associated with running many small VM-types, that is assumed to be included in the capacity figure. The price of a VM, p_i, is defined as all possible costs of maintaining a VM from a customer perspective, regardless if this is specified costs from a public provider or costs somehow determined for using a private infrastructure. In order to find the optimal set, we formulate the problem as an integer programming problem.

3.2 Deciding on transitioning policy

The knowledge about an optimal set of VMs for a certain capacity only gives a set of VMs that minimizes the cost, without acknowledging the overhead costs of altering the configuration. Therefore, we need to calculate the costs of performing this reconfiguration, in order to decide whether or not it is beneficial to enforce the reconfiguration despite the overhead costs. To calculate the reconfiguration costs, we first specify how to move from the current set to the optimized set. The transition policy defines a plan for changing the resource set from its current setup S to its new target configuration $\hat{S}$. Depending on application type and cloud platform, the transition can be done in several ways, e.g., by shutting down the VM, copying the state to the new VM, and starting the new VM; or by keeping both VMs running while new VMs boot and become ready to serve; or a combination of both by shutting down some VMs and keeping some other running. Transitions also can be carried out as a single action or they can be done through a series of consecutive actions, e.g., first start a VM and then shut down extra VMs one by one. For either alternative, there is an associated cost, called reconfiguration cost.

In our experiments, the transition policy is based on the notion of always maintaining the required capacity by keeping all instances running, to restrict the performance loss during transition. In order to do that, we start all new instances while maintaining the previous configuration running. Only after the new instances are launched and ready to serve, we shut down the extra instances from the previous configuration.

3.3 Calculating the reconfiguration costs

Reconfigurations, no matter the transition policy chosen, require changes in the resource set and a number of VM startups and shutdowns. Although cloud VM instances are expected to be very rapidly provisioned, the total time before they are ready to serve the application may be substantial. In our case, this time is the reconfiguration time. The minimum reconfiguration time is the time it takes for booting the VM and registering it to the load balancer until it becomes available. This time can be much longer when we add the time for application specific configurations, copying the data files and startups. For longer reconfigurations, the cost of preserving the performance increases, e.g., if a reconfiguration action requires copying a large database to a new VM, both VMs need to be up while the database is being copied, leading to extra payments for having two VMs in use. During the reconfiguration time a performance impact can be expected and this impact is associated with a cost. This cost can be defined in terms of performance degradations, SLA violations or simply the additional costs required to preserve the QoS during transition time. We calculate these reconfiguration costs as follows:

$$R = C_{extra} \times repacking\ time$$

where C_{extra} is the overhead cost associated with the extra allocation needed to preserve the performance during the repacking time.

3.4 Decision making

Deciding when to reconfigure is based on a cost-benefit analysis that depends on the following factors:

1. The difference between the current configuration's cost and the optimal configuration's cost.
2. The cost of reconfiguration.
3. The time period that we expect the new configuration to last.

Each of these factors is part of a decision function that weights the potential benefits of reconfiguration against the reconfiguration cost and the decision evaluates to true if the condition in Equation (1) is met.
Here $\hat{P}$ and P are respectively the runtime costs of the new and current configuration, d is the expected duration, or stability interval, which is the time we expect the new configuration be durable. If this duration is long, even expensive reconfigurations are worth performing. R is the reconfiguration cost and K is the gain factor which defines the minimum benefit expected for performing the reconfiguration. K represents hidden costs different from explicit reconfiguration costs, e.g., additional cost due to software licenses, or additional human resources for maintenance. These costs are not easily measured, but still need to be considered. We address these costs by assuming that a human manager configures the controller based on knowledge about such costs, gained from experience.
As it can be seen in (1), for large values of d, $\hat{P} \times d$ becomes large enough, that R can be neglected. This means that if we expect the new optimized configuration to last long enough, the reconfiguration costs can be neglected and the repacking controller can proceed with reconfiguration of the resource set. Although for large values of d, the maximum cost saved is bounded by $1 - (\hat{P}/P)$.

4 Autoscaling strategies

Autoscalers adapt the resource set to fast and transient changes in load. Different autoscalers have different VM selection policies and decide based on different inputs. Hence they propose different resource sets, which affect the decisions and costs of repacking. The autoscaler calculates the required

Algorithm 1 Repacking Algorithm

Input: Required Capacity C at time t; Current Resource Set (S), with associated cost P.

Output: Decision on Repacking

```
1:  Find new optimal Resource Set, $\hat{S}$.
2:  Calculate cost ($\hat{P}$)
3:  if $\hat{P} < P$ then
4:      Select a Transition Policy
5:      Calculate Reconfiguration Cost:
        $R = C_{extra} \times repacking\ time.$
6:  end if
7:  if $1 - [(\hat{P} \times d + R)/(P \times d)] > k$ then
8:      Reconfigure to $\hat{S}$
9:      return;
10: else
11:     Repacking is not feasible
12:     return;
13: end if
```

capacity either by taking *total load* or $\Delta load = total\ load - current\ capacity$ as input.

Additionally, the policy for selecting VM types to provide the capacity can also be done at least in two sensible ways. It can either select the *optimal set of VMs*, or select a specific VM type preferred by the application or specific to a workload behavior, we call this prefered VM a *base instance*.

Repacking can be applied on autoscalers using $\Delta load$ as input. So using a repacking approach, two new strategies are introduced, $S_{delta\_Repacked}$ and $S_{base\_Repacked}$. This results in 5 different autoscaling strategies in total, that are presented as follows:

1. S_{delta}:
S_{delta} calculates an optimal resource set for providing capacity corresponding to $\Delta load$. Some applications are performance sensitive and may require quick adaptations to changes and their priority is to maintain performance even by paying higher rental costs. S_{delta} is an autoscaling strategy suitable for this kind of applications. It does not perform reconfigurations, since it only adds (or removes) the VMs to the previous allocation. However, if the workload increases in small steps, S_{delta} allocates one more small VM for each time the load changes. The output of S_{delta} over time is therefore typically a resource

set comprises of a set of small instances. Although each addition is optimal with respect to $\Delta load$, it is still a sub-optimal set with respect to the *total load.*

2. S_{base}:
The S_{base} strategy similarly takes $\Delta load$ as input for scaling the capacity. However, contrary to the first strategy, S_{base} does not adjust to the exact optimal extra capacity, i.e., a small VM for a small change, but it adds a VM of a predefined size, *base instance*, that a priori, has been judged to be a good size for that specific workload characteristics, such as volume and its oscillations in a specific time span. The *base instance* does not necessarily fit the capacity perfectly, as it may overprovision a little to avoid the frequent additions and removals that lead to a set of small expensive VMs in S_{delta}. So *base instance* can be selected by considering a tradeoff between overprovisioning cost and the cost due to cost inefficiency of smaller VMs. Another reason for using S_{base} is that a VM type may be more suitable for a certain application, e.g., a compute intensive application benefits from a high CPU VM type. The resource set shaped by this strategy is homogeneous consisting of a set of *base instances* for the duration that workload volume and its changes are within a specific range.

3. S_{Full}:
The previous two strategies take $\Delta load$ as input and find the appropriate extra VMs to satisfy $\Delta load$. In both cases the previous resource set remains unchanged and the new VMs are added to keep up with the changes in load. S_{Full}, on the contrary, uses *total load* to find the optimal set to serve the application demand. By considering *total load*, S_{Full} not only adds extra capacity but it may also replace VMs with more cost efficient ones, during any scaling decision. In this way, S_{Full} maintains the resource set optimal during application runtime. However, replacements may require frequent shutdowns and start ups that are costly and failure prone. The cost of running the application with this strategy is the sum of the resource set cost plus the cost of reconfigurations at each timestep.

The drawback of S_{Full} is that it blindly performs the reconfiguration as long as the new resource set is not optimal, without taking reconfiguration cost into account. We include S_{Full} as a comparison to show how doing a cost benefit analysis optimizes the number of reconfigurations as they are considered to be costly. We investigate the impact of smarter reconfigurations on the total cost against blind reconfigurations by comparing our result against S_{Full}.

4. $S_{base\_Repacked}$ and $S_{delta\_Repacked}$:
Repacking is applicable on autoscalers with $\Delta load$ as input. Hence, we applied our repacking model on the resource sets suggested by S_{delta} and S_{base} that leads to two new scaling strategies called $S_{delta\_Repacked}$ and $S_{base\_Repacked}$ repacking. The repacking model uses each of the resource sets proposed by S_{delta} and S_{base} to investigate which reconfigurations are required to reshape the set to optimal and to discover the cost of these reconfigurations. This information is used to perform a cost benefit analysis to evaluate the worthiness of the repacking. If the evaluation is positive, the previous resource set is replaced by the new optimal one. The reconfiguration actions for S_{delta} are mostly exchanging smaller instances with larger ones, whereas for S_{base} the actions involve replacing the set with an optimal one formed of different types of instances, instead of the original set of a single VM type.

Table 1 summarizes the autoscaling strategies employed, their inputs and specifications.

Table 1: *Autoscaling strategies*

Scaling strategy	**Input**	**VM selection policy**	**Repacking**
S_{delta}	$\Delta load$	optimal set	never
S_{base}	$\Delta load$	*base instance*	never
S_{Full}	*total load*	optimal set	always
$S_{delta\_Repacked}$	*total load*	optimal set	when beneficial
$S_{base\_Repacked}$	*total load*	optimal set	when beneficial

5 Repacking and workloads

The need and effects of repacking of VMs are different for different workload patterns. Mao and Humphrey [7] characterize four types of workloads for cloud environments as *Stable*, *Increasing (Growing)*, *Seasonal (Cycle/Bursting)* and *On-and-Off*. Each of these patterns represents a specific application or a scenario.
Stable and **On-and-Off workloads**: The behavior of these workloads are similar to batch workloads. They have a definite lifetime and resource requirements, e.g., many repetitive transactions to a database, with some heavy computational work for each. They are also active for a short period, *on-and-of*, or a long period of time, *Stable*, but their load is constant and not changing overtime.
Increasing workloads: Workloads of this category are often seen in companies that have high growth trends. Social networking services such as Facebook,

Twitter, or file sharing applications such as Dropbox are good samples of *increasing* workloads. The growth trend for these websites shows a common pattern, as they all start with slow growth followed by small oscillations that become larger as service become popular.

Seasonal (Cycle/Bursting) workloads: The increase in load in seasonal workloads occurs in patterns that are repeating themselves, e.g., daily, weekly or monthly periods. These increases can be based on predictable events (in time and possibly in magnitude).

Neither of the *Stable* and *On-and-Off* workloads is of particular relevance to repacking of VM sets as the optimal set can be chosen from start and remain optimal since there are no load variations during the execution. Therefore, we focus on the *increasing* and *seasonal* workloads as they have load variations where series of autoscaling actions over time may lead to suboptimal VM set.

6 Performance evaluation

6.1 Experiment setup

We evaluate our repacking approach by simulating a cloud with 6 types of VMs. In order to study the effect of repacking for different price/performance ratios between VM-types we consider two application types, A and B. Each application benefit differently from the different types of VMs, e.g., by replacing 2 small VMs with a medium VM, application A can serve 1.2 times more request for the same price, while this number for application B is 1.4 times more request for the same reconfiguration.

For clarity, we have assumed that allocated resources can be used and paid for any period of time and not on, e.g., a per-hour basis. This scenario is most relevant to a private cloud but the approach can equally well be applied to infrastructures with hourly payment schemes, although the time for repacking should then be aligned with payment periods to make full use of resources already paid for.

Table 2 shows the resource utilization for each application on each VM type. In our evaluations we consider seasonal and increasing workload patterns. We perform the evaluation by generating two synthetic workloads of each type, as shown in Figure 1. Our 4 workloads were different in volume and duration with 168, 182, 215 and 338 hours length.
Since the time required for actuating a reconfiguration contributes with overhead costs that impacts the decision of when it is beneficial to perform reconfiguration, we include two different reconfiguration durations, 4 minutes and 20 minutes. We also set the gain factor $K = 1$ % which implies that the expected

Table 2: *Server configurations and their prices*

Application	**VM Type**	**Price($/min)**	**Capacity(req/min)**
Application A	small1	0.003	500
	small2	0.006	1197
	med1	0.012	2867
	med2	0.024	6868
	large1	0.048	16449
	large2	0.090	39396
Application B	small1	0.003	500
	small2	0.006	1310
	med1	0.012	3434
	med2	0.024	9000
	large1	0.048	23588
	large2	0.090	61819

benefit should be higher than 1% in order for repacking to be performed. Notably, 1% may seem like a small number but it should be remarked that it is with respect to total infrastructure cost.
For each application runtime, we measure the following parameters:

1. **Total cost of running the application.** The total cost is the aggregated resource utilization cost, from start to the end. We compare the total cost of an application when using different scaling strategies. This helps us to reason about the cost efficiency of each of them.

2. **Number of reconfigurations performed.** The number of reconfigurations during the application lifetime provides insight into the effects of reconfiguration cost and stability interval of the reconfiguration. It also shows how the cost benefit analysis filters the unnecessary reconfigurations and result a reduction in total cost.

VM types are each associated with a price p_i for resources consumed during a time unit. Each VM type serves a maximum capacity C_i^A and C_i^B for application A and B respectively. *Total load* is measured as the aggregated load at time t where $\Delta load = total\ load_t - capacity_{t-1}$.
We evaluate five strategies, S_{delta}, $S_{delta\_Repacked}$, S_{base}, $S_{base\_Repacked}$ and S_{Full}. We compare the total cost of running an application for each of these five strategies. For the scaling strategies that are selecting *optimal set* of VMs, i.e, $S_{delta\_Repacked}$, $S_{base\_Repacked}$ and S_{Full}, we formulate the problem of finding the optimal set as an Integer Linear Programming (ILP) model and used Gurobi solver to solve this problem. The result is an instance set, that represents the VMs with a sufficient capacity and minimum cost.

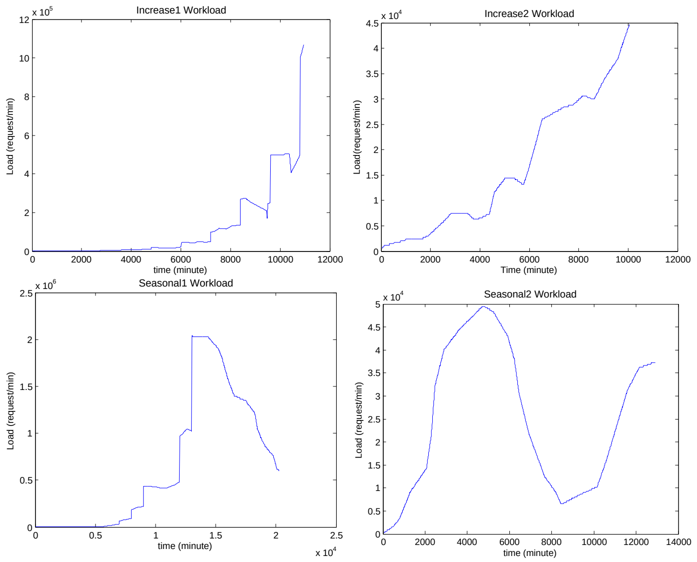


Figure 1: The workloads used in experiments

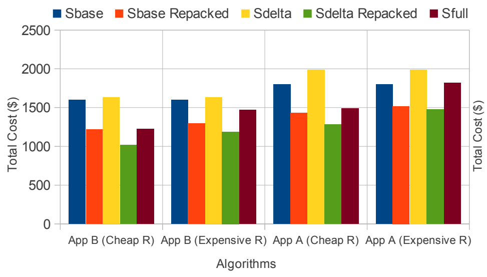
6.2 Repacking and scaling algorithms

To demonstrate the effect of repacking, we compare the autoscaling strategies presented in Section 4 when using our proposed repacking method $S_{delta\_Repacked}$ and $S_{base\_Repacked}$ against the same strategies but without repacking (S_{delta}, S_{base}, S_{Full}).

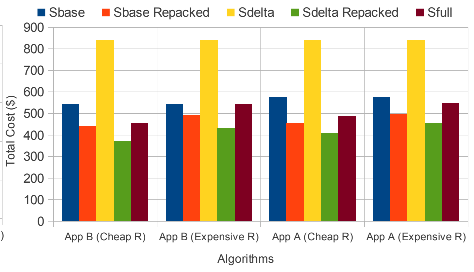
Table 3 and Figure 2 show the total cost of running applications A and B for four workloads. In Figure 2 each set of bars compares the total cost of running each application, on *Y axis*, for five autoscaling strategies and two different reconfiguration costs (*Cheap R* and *Expensive R*). The first and third set of bars show the result of repacking when the reconfiguration cost is cheap whereas the second and fourth set present the result for expensive reconfigurations. The following are observations after analyzing these 5 autoscaling strategies:

1. Comparison of non-repacked scaling mechanisms against their repacked counterparts (S_{delta} with $S_{delta\_Repacked}$ and S_{base} with $S_{base\_Repacked}$) shows that the total cost is reduced when repacking is applied.

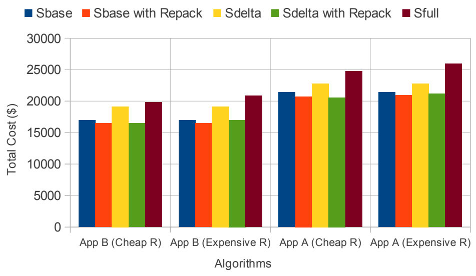
2. Comparison of repacking strategies ($S_{delta\_Repacked}$ and $S_{base\_Repacked}$) against S_{Full} shows that the total cost obtained when applying the repacking strategy is lower than the cost of S_{Full}. Lower cost is a result of performing reconfigurations at the appropriate time avoiding unnecessary costly reconfigurations thus reducing the total cost of running the application.



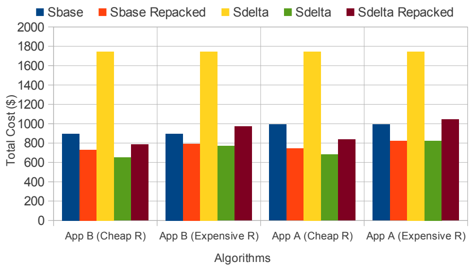
(a) Increase1 workload



(b) Increase2 workload



(c) Seasonal1 workload



(d) Seasonal2 workload

Figure 2: Comparison between total costs of running applications resulted by different scaling algorithms and different workloads when $d = 1000$.

3. $S_{delta\_Repacked}$ (green bars) outperforms all other scaling strategies, producing the lowest total cost for running the applications. S_{delta} by itself is not an efficient autoscaler but it generates a resource set comprised of various VM types that increases the possibility of reducing reconfiguration costs. Repacking tries to keep common VMs in the VM set, adding extra VMs of other types only when they are not available. When repacking S_{delta}, reconfiguration costs are low because it is very probable to reuse an already started VM. This reduces the total cost after repacking.

4. $S_{base\_Repacked}$ (red bars) shows the second best result among the scaling strategies tested. S_{base} produces a VM set of mostly the same type of VMs, so transitioning to an optimal set requires starting several VMs of other types. This leads to higher reconfiguration costs making this strategy less efficient than $S_{delta\_Repacked}$.

5. S_{delta} (yellow bars) is the most expensive and least cost efficient strategy among all. The reason is that S_{delta} produces a resource set that mostly consists of a large number of small instances with high *price/performance* ratios.

6. S_{base} is a more efficient strategy than S_{delta} since it is less sensitive to small spikes in load, making it a more stable strategy. S_{base} can handle those spikes by previously over-provisioned resource sets. However, for this to happen, *base instance* must be correctly determined with respect to workload volume and oscillations, which by itself is a non-trivial problem.

Table 3: *Total cost (\$) of running the servers with different scaling models and different workloads when* $d = 1000$

Workload type	**Reconfig cost**	**Cheap**	**Expensive**	**Cheap**	**Expensive**
	Scaling strategy	App A		App B	
Increase1	S_{base}	1797.62	1797.62	1598.3	1598.3
	$S_{base\_Repacked}$	1428.86	1518.42	1218.03	1296.65
	S_{delta}	1989.62	1989.62	1630.12	1630.12
	$S_{delta\_Repacked}$	1283.17	1475.6	1020.55	1189.86
	S_{Full}	1488.69	1819.13	1224.64	1472.33
Increase2	S_{base}	576.27	576.27	544.86	544.86
	$S_{base\_Repacked}$	456.84	495.73	443.18	490.94
	S_{delta}	838.31	838.31	838.29	838.29
	$S_{delta\_Repacked}$	407.32	455.56	373.43	433.09
	S_{Full}	488.79	545.9	454.06	542.85
Seasonal1	S_{base}	21393.74	21393.74	16961.23	16961.23
	$S_{base\_Repacked}$	20732.84	20969.79	16502.5	16517.3
	S_{delta}	22801.98	22801.98	19152.16	19152.16
	$S_{delta\_Repacked}$	20579.89	21193.29	16476.5	17003.44
	S_{Full}	24740.29	25936.8	19839.8	20857.36
Seasonal2	S_{base}	993.19	993.19	894.91	894.91
	$S_{base\_Repacked}$	743.83	820.75	730.58	791.78
	S_{delta}	1745.86	1745.86	1744.65	1744.65
	$S_{delta\_Repacked}$	681.06	824.12	645.35	769.06
	S_{Full}	835.6	1043.87	784.48	970.67

We also studied the case when an application owner wants to use only one instance type. In this case a new *base instance* is used for repacking instead of repacking to optimal. This strategy results in expensive reconfiguration costs, as the entire resource set should be replaced with the new resource set that only uses the new *base instance*. However, repacking still produces a benefit since it performs a cost-benefit analysis before doing the reconfigurations. In this case the number of reconfigurations performed is low due to the high reconfiguration costs. However, from a pure cost-perspective this approach is not competitive.

6.3 Reconfiguration cost and stability intervals

Repacking decisions are closely dependent on the reconfiguration cost. This extra cost needs to be carefully considered before deciding to repack to evaluate if repacking is actually beneficial. We defined this cost as a function of reconfiguration time and it is the time required for the new servers to become available, or the time that both old and new resource sets are concurrently alive in order to preserve performance. We studied the behavior of the repacking model when the reconfiguration cost is cheap and expensive.

Before repacking we need to estimate the expected stability interval, d, of the new configuration. In order to show the effects of stability intervals on reconfiguration numbers, we assumed that there is uncertainty in predicting stability intervals. In this experiment we have assumed a rough prediction of $d = 180$ minutes for *Increase1, Seasonal1*, and *Seasonal2* workloads, and $d = 100$ minutes for *Increase2* workload. The $d = 100$ minutes and $d = 180$ minutes in Table 4 highlight the effects of short stability intervals on repacking decisions with expensive reconfigurations by showing a decrease in number of reconfigurations. We also have a more accurate estimation of $d = 1000$ minutes for our workloads, which shows the behavior of the repacking model for long stability intervals. In our case, we set the values of d by observing the workload pattern beforehand, although this value can be forecasted using statistical models such as ARIMA [11]. With this setup we make the following observations.

1. Increasing the reconfiguration cost increases the total cost of running the application in S_{Full}, $S_{base\_Repacked}$ and $S_{delta\_Repacked}$.

2. Doing expensive reconfigurations for a short stability interval (d) is not reasonable since the next reconfiguration takes place before the overhead cost of a previous one can be amortized. As presented in Table 4, when reconfiguration cost is expensive the repacking algorithm decides to perform 13 reconfigurations less compared to when reconfigurations are cheap in $Increase1$ workload with $d = 180$. This argument is also valid for *Seasonal1*, *Seasonal2*, and *Increase2* workloads, with 20, 12, and 10 less reconfigurations respectively.

3. If a reconfiguration lasts long enough even costly reconfigurations can be beneficial. In this case the number of reconfigurations are almost equal for both cheap and expensive scenarios.

Our experiments show that for cheaper reconfigurations more reconfiguration actions are triggered and the resource set is more frequently repacked to the optimal with a low reconfiguration cost. In contrast, for high reconfiguration

Table 4: *Number of reconfigurations with respect to different stability intervals (d)*

Workload	**d=180 min**		**d=1000 min**	
Reconfig cost	Cheap	Expensive	Cheap	Expensive
Increase1	46	33	46	46
Seasonal1	87	67	89	89
Seasonal2	50	38	52	49
Workload	**d=100 min**		**d=1000 min**	
Reconfig cost	Cheap	Expensive	Cheap	Expensive
Increase2	22	12	23	23

Table 5: *Total cost of running the application for different Stability interval, d values*

Workload		**Cheap**		**Expensive**	
	Stability interval	d=180	d=1000	d=180	d=1000
Increase1	Number of reconfigs	46	46	33	46
	Total cost	1020.55	1020.55	1209.32	1189.86
Seasonal1	Number of reconfigs	87	89	67	89
	Total cost	16490.54	16476.5	17017.66	17003.44
Seasonal2	Number of reconfigs	50	52	38	49
	Total cost	646.38	645.35	857.48	824.12
Workload		**Cheap**		**Expensive**	
	Stability interval	d=100	d=1000	d=100	d=1000
Increase2	Number of reconfigs	22	23	12	23
	Total cost	374.01	373.43	468.79	433.09

costs some reconfigurations are avoided since the cost benefit analysis does not show any benefit for doing costly reconfigurations.

Table 5 shows the effect of the number of beneficial reconfigurations on the total cost of running the application. A reduced number of beneficial reconfigurations is either due to a short stability interval or to a high reconfiguration cost and it leads to an overall higher resource cost, the more reconfigurations, the more opportunities for tuning the resource set to optimal. However, the number of reconfigurations is not the only factor for reaching an optimal allocation. A more important factor for total cost reduction is the difference between the optimal cost and the cost for the old VM set for each reconfiguration opportunity.

6.4 Price/performance ratio

When running our two applications, *App A* and *App B*, we can choose from 6 VM types. Each VM$_i$ can serve application A with performance C_i^A and application B with performance C_i^B at a cost p_i. A larger i represents a larger VM so that $C_{i+1} > C_i$. We use the *price/performance* ratio (p_i/C_i) as metric for cost efficiency of VM$_i$ for the specific application. The repacking assumption is that the *price/performance* ratio of larger VMs is lower than for smaller ones, so repacking smaller instances becomes reasonable.

To understand which *price/performance* ratios make repacking of interest, in our experiments, we assume that our applications are different in relative performance ratio according to VM sizes. So, for the same p_{i+1}/p_i, $C_{i+1}^A/C_i^A < C_{i+1}^B/C_i^B$, i.e., for the same price, larger VMs are capable of serving more requests for application B in comparison to application A. Table 2 shows the *price* and *capacity* of different VMs of each application used in our experiments.

With these parameters, repacking is more beneficial for applications with a larger difference between p_{i+1}/C_{i+1} and p_i/C_i, in our case application B. As seen in Table 6, the total cost of running application A with *Increase1* workload when the reconfiguration costs are cheap is reduced by 35.51%, while this number is 37.39% for application B that has a lower *price/performance* ratio. This reduction is 25.84% for application A, while it is 27.01% for application B that has high reconfiguration costs. The same pattern can be seen when the applications encounter other workload types or have high reconfiguration costs. The reason is that for applications with low *price/performance* ratios, larger instances are capable of serving more requests for a lower cost, which is a good motivation for repacking the instances to larger VMs in order to reduce the total cost of running the application.

Table 6: *Improvement percentage achieved by using repacking for applications A and B*

Workload	**Cheap R**		**Expensive R**	
	App A	**App B**	**App A**	**App B**
Increase1	35.51 %	37.39 %	25.84%	27.01%
Increase2	51.41%	55.45%	45.66%	48.34%
Seasonal1	9.74%	13.97%	7.06%	11.22%
Seasonal2	60.99%	63.04%	52.8%	55.92%

7 Related work

There are several contributions that relate to our work. The first group of related studies focus on capacity autoscaling in clouds [5,6,8,10,12,13]. Their objective is typically to adjust the allocated capacity to demand so that the required performance is provided, and their main concern is to predict the capacity as accurate as possible to ensure that the capacity is available when needed. Methods such as, static threshold based controllers [8,13], control theory [14,15], queuing theory [5, 12], and time series analysis of workloads [9, 16] are used to handle autscaling problem in clouds.

Another group of related studies focuses on cost efficient resource provisioning in cloud environments. Cost efficiency in clouds can be discussed from an infrastructure provider or from an application owner point of view. The difference is that infrastructure providers are more concerned with high resource utilization and energy efficiency of their infrastructure, so they try to avoid overprovisioning as much as possible by efficient consolidation strategies [17,18] or overbooking policies [19, 20]. On the other hand, application owners are concerned with the performance of their applications and budget constraints, so their focus is on cost aware scheduling strategies [4, 21, 22] and smart resource acquisition [7] to handle the job with the minimum budget. However, in all the aforementioned works, planning the capacity is normally performed solely based on load, regardless of considering an optimal combination of VM types. To the best of our knowledge, only few studies taking VM sizes and their performances into account but then mostly for the optimal placement of VMs [23–25] rather than autoscaling.

Runtime reconfiguration techniques and measuring their associated costs is also a topic of interest when studying repacking. Jung et al., [26], proposes an adaptation engine for runtime reconfiguration in multi tier applications. They evaluate the impact of 5 reconfiguration actions on application response time and developed a middleware to generate cost-aware adaptation actions. However, they approached the problem of on-demand reconfiguration from an infrastructure provider perspective. Although the overall idea of reconfiguration is similar for both infrastructure providers and application owners, the target problem is different. Infrastructure providers often perform reconfigurations to ensure availability and isolation of demand fluctuations in co-located VMs, whereas application owners employ reconfigurations to reduce cost and increase performance.

Sharma et al, [3], introduce a cost aware provisioning system that takes into account price differentials of server types in order to minimize the rental cost of the application. They considered cost aware reconfiguration of resources as part of the autoscaling mechanism. While this is the closest study to our work, a main difference is that we decouple reconfiguration from autoscaling so that more freedom is given to the application owner when choosing autoscaler. The result is that application owners can choose the autoscaler that better fits the application's needs while at the same time benefiting from the advantages of repacking.

8 Conclusion

In this paper we have shown that combining the benefits of *vertical* and *horizontal* elasticity to scale in terms of both the number and the size of VMs increases the cost efficiency of the resource set used to serve an application. We propose a cost-benefit based approach called *repacking*, which takes cost and stability of a reconfiguration into account to determine the appropriate trade-off between horizontal and vertical scaling and acts accordingly. The goal of the repacking method is to find a configuration of resources that minimizes the resource utilization cost, given a current configuration, a number of different VM-types with different price/performance characteristics, a cost-function for performing repacking, and an expected time (duration) for which the new configuration is expected to be used.

Through experimental evaluations we have shown that by performing a cost-benefit analysis we can decide when and how to replace a non-optimal set by a new optimal set, and that decision can reduce the total cost of resource utilization during the application's lifetime. Our results show a range of 7% up to 60% cost saving in total resource utilization cost of our sample applications and workloads.

We also show that different applications benefit differently from the repacking approach. Applications with larger differences in *price/performance* ratio among their VM types are the ones that benefit most. Moreover, we studied the effects of cheap and costly reconfigurations on repacking behavior. Our results show that costly reconfigurations become reasonable to perform when the expected stability of the reconfiguration is long enough to amortize the overhead cost of reconfiguration. Hence, our approach automatically avoids unnecessary reconfigurations if a short stability interval for that reconfiguration is expected.

9 Acknowledgement

Financial support for this work is provided in part by the European Commissions Seventh Framework Programme ([FP7/ 2001 -2013]) under grant agreement number 257115, OPTIMIS and Swedish Research Council (VR) under contract number C0590801 for the project Cloud Control.

References

[1] M. Sedaghat, F. Hernández, and E. Elmroth, "Unifying cloud management: Towards overall governance of business level objectives," in *11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pp. 591–597, IEEE, 2011.

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

[3] U. Sharma, P. Shenoy, S. Sahu, and A. Shaikh, "Kingfisher: A system for elastic cost-aware provisioning in the cloud," *Dept. of CS, UMASS, Tech. Rep. UM-CS-2010-005*, 2010.

[4] M. Salehi and R. Buyya, "Adapting market-oriented scheduling policies for cloud computing," *Algorithms and Architectures for Parallel Processing*, pp. 351–362, 2010.

[5] A. Ali-Eldin, J. Tordsson, and E. Elmroth, "An adaptive hybrid elasticity controller for cloud infrastructures," in *Network Operations and Management Symposium (NOMS), 2012 IEEE*, pp. 204–212, IEEE, 2012.

[6] T. Lorido-Botrán, J. Miguel-Alonso, and J. A. Lozano, "Auto-scaling techniques for elastic applications in cloud environments," tech. rep., Department of Computer Architecture and Technology, UPV/EHU, 2012.

[7] M. Mao and M. Humphrey, "Auto-scaling to minimize cost and meet application deadlines in cloud workflows," in *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis*, (New York, NY, USA), pp. 49:1–49:12, 2011.

[8] "Rightscale cloud management. http://www.rightscale.com/, 2012."

[9] Z. Gong, X. Gu, and J. Wilkes, "Press: Predictive elastic resource scaling for cloud systems," in *International Conference on Network and Service Management (CNSM), 2010*, pp. 9–16, IEEE, 2010.

[10] H. Ghanbari, B. Simmons, M. Litoiu, C. Barna, and G. Iszlai, "Optimal autoscaling in a IaaS cloud," in *Proceedings of the 9th international conference on Autonomic computing*, pp. 173–178, ACM, 2012.

[11] J. M. Tirado, D. Higuero, F. Isaila, and J. Carretero, "Multi-model prediction for enhancing content locality in elastic server infrastructures," in *18th International Conference on High Performance Computing (HiPC)*, pp. 1–9, IEEE, 2011.

[12] R. Chi, Z. Qian, and S. Lu, "A game theoretical method for auto-scaling of multi-tiers web applications in cloud," in *Proceedings of the Fourth Asia-Pacific Symposium on Internetware*, Internetware '12, (New York, NY, USA), pp. 3:1–3:10, ACM, 2012.

[13] M. Maurer, I. Brandic, and R. Sakellariou, "Enacting slas in clouds using rules," *Euro-Par 2011 Parallel Processing*, pp. 455–466, 2011.

[14] P. Padala, K.-Y. Hou, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, and A. Merchant, "Automated control of multiple virtualized resources," in *Proceedings of the 4th ACM European conference on Computer systems*, pp. 13–26, ACM, 2009.

[15] T. Patikirikorala and A. Colman, "Feedback controllers in the cloud," *Swinburne University*, 2011.

[16] J. Kupferman, J. Silverman, P. Jara, and J. Browne, "Scaling into the cloud," *CS270-Advanced Operating Systems*, 2009.

[17] M. Mazzucco, D. Dyachuk, and R. Deters, "Maximizing cloud providers' revenues via energy aware allocation policies," in *3rd International Conference on Cloud Computing (CLOUD), 2010 IEEE*, pp. 131–138, IEEE, 2010.

[18] I. Goiri, J. Guitart, and J. Torres, "Characterizing cloud federation for enhancing providers' profit," in *3rd International Conference on Cloud Computing (CLOUD), 2010 IEEE*, pp. 123–130, IEEE, 2010.

[19] B. Urgaonkar, P. Shenoy, and T. Roscoe, "Resource overbooking and application profiling in shared hosting platforms," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 239–254, 2002.

[20] D. Breitgand, Z. Dubitzky, A. Epstein, A. Glikson, and I. Shapira, "Sla-aware resource over-commit in an iaas cloud," in *8th international conference on systems virtualiztion management (svm) Network and service management (cnsm), 2012*, pp. 73–81, Oct.

[21] T. N. B. Duong, X. Li, R. Goh, X. Tang, and W. Cai, "Qos-aware revenue-cost optimization for latency-sensitive services in iaas clouds," in *16th*

International Symposium on Distributed Simulation and Real Time Applications (DS-RT), 2012 IEEE/ACM, pp. 11–18, Oct.

[22] M. Zhu, Q. Wu, and Y. Zhao, "A cost-effective scheduling algorithm for scientific workflows in clouds," in *31st International on Performance Computing and Communications Conference (IPCCC), 2012 IEEE*, pp. 256–265, Dec.

[23] W. Li, J. Tordsson, and E. Elmroth, "Modeling for dynamic cloud scheduling via migration of virtual machines," in *Proceedings of Third International Conference on Cloud Computing Technology and Science*, CLOUDCOM '11, pp. 163–171, 2011.

[24] W. Li, J. Tordsson, and E. Elmroth, "Virtual machine placement for predictable and time-constrained peak loads," in *Proceedings of the 8th international conference on Economics of Grids, Clouds, Systems, and Services*, GECON'11, (Berlin, Heidelberg), pp. 120–134, Springer-Verlag, 2012.

[25] J. Tordsson, R. S. Montero, R. Moreno-Vozmediano, and I. M. Llorente, "Cloud brokering mechanisms for optimized placement of virtual machines across multiple providers," *Future Gener. Comput. Syst.*, vol. 28, pp. 358–367, Feb. 2012.

[26] G. Jung, K. Joshi, M. Hiltunen, R. Schlichting, and C. Pu, "A cost-sensitive adaptation engine for server consolidation of multitier applications," *Middleware 2009*, pp. 163–183, 2009.

Paper VI

Die-Hard: Reliable Scheduling to Survive Correlated Failures in Cloud Datacenters

M. Sedaghat, E. Wadbro, J. Wilkes, S. De Luna, O. Seleznjev, E. Elmroth

Submitted for publication.

Die-Hard: Reliable Scheduling to Survive Correlated Failures in Cloud Data Centers[*]

Mina Sedaghat[†], Eddie Wadbro[†], John Wilkes[‡],
Sara De Luna[§], Oleg Seleznjev[§], Erik Elmroth[†]

Abstract

In large scale data centers, a single fault can lead to correlated failures of several physical machines and the tasks running on them, all at once. Such correlated failures can severely damage the reliability of a service or a job running on the failed hardware. To mitigate the impacts of correlated failures, data centers must identify failure domains that are exposed to common failure sources and determine how to replicate and schedule extra tasks to guarantee the desired reliability.

This paper models the impact of stochastic and correlated failures on job reliability in a data center. We specifically focus on correlated failures caused by power outages or failures of network components, on jobs running multiple replicas of identical tasks. We present a statistical reliability model and an approximation technique for computing a job's reliability in the presence of correlated failures.

In addition, we address the problem of scheduling a job with reliability constraints. We formulate the scheduling problem as an optimization problem, with the aim being to minimize the number of concurrent task failures due to a single fault and to maintain the desired reliability with the minimum number of extra tasks. We present a scheduling algorithm that approximates a minimum number of required tasks and a placement to guarantee a desired job reliability. We study the efficiency of our algorithm using an analytical approach and by simulating a cluster with different failure sources and reliabilities. The results show that the algorithm can effectively approximate the minimum number of extra tasks required to guarantee the job's reliability.

[*]This paper is submitted for publication.

[†]Dept. of Computing Science, Umeå University, email: {mina, eddiew, elmroth}@cs.umu.se

[‡]Google Inc., johnwilkes@google.com

[§]Dept. of Mathematics and Mathematical Statistics, Umeå University, SE-901 87 Umeå, Sweden, email: {sara.de.luna, oleg.seleznjev}@math.umu.se

Keywords: Cloud computing; Scheduling; Reliability; Fault tolerance; Correlated failures.

1 Introduction

Data centers achieve high reliability through the use of failure tolerant hardware, network equipment, and architectures or via adopting sophisticated management solutions such as replication and recovery techniques. Many of these techniques try to achieve reliability for a single individual component such as a server or a software component, rather than providing overall reliability for jobs or services. However, it is not generally valid to assume that servers fail independently and component failures are uncorrelated. Correlated failures such as failures due to power outages or network component failure are rare [1] but have significant effects on system reliability [2–6]. Ignoring the impact of correlated failures can cause reliability to be overestimated by at least two orders of magnitude [2].

In this work we present a statistical model for job reliability in a cloud data center, in the presence of stochastic and correlated failures. The model quantifies the impacts of correlated failures on the overall reliability of a job comprising multiple identical tasks. The job reliability is defined as the probability that at least a minimum number of tasks will continue running throughout the job's runtime. We specifically focus on correlated failures caused by power outages and failures of network components. In our model, power nodes and network components have different failure rates and their failures affect different sets of servers, known as *failure domains*. The problem becomes complicated because different failure events may be associated with different failure domains. For example, the servers that are affected by a power outage will not necessarily be the ones that are affected by a network component failure [7].

Delivering job reliability cannot be considered independently from job scheduling because job reliability is highly dependent on the placement of the tasks over different failure domains (e.g. different racks or power failure domains). To achieve job reliability, scheduling and allocation decisions should take into account the probabilities of failure for different servers, racks, and other potential failure domains, along with the impact of failures in these domains on the job's reliability.

Using the proposed reliability model, we introduce a scheduling algorithm to guarantee a given level of reliability for a particular job. The objective of the algorithm is to guarantee an overall reliability for a job while using the minimum necessary number of extra tasks (replicas), and to schedule the job across multiple different failure domains. Scheduling tasks over different failure domains makes jobs robust against the negative impact of a single failure, while running extra tasks enables the job to compensate for losses and deliver the targeted minimum reliability during recovery.

The scheduling algorithm approximates a minimum number of required replicas and selects a subset of machines in the cluster to run replicas of the tasks. Note that replicating a task within a failure domain cannot prevent it from being affected by correlated failures, no matter how much redundancy is introduced. Similarly, spreading allocations over different failure domains alone is not sufficient to guarantee that the targeted minimum reliability will be achieved. Therefore, a combination of good placement and replication is required to ensure high overall reliability.

The contributions of this paper are:

1. A statistical reliability model for a job running multiple identical tasks, deployed in a cloud data center. The model captures the impacts of stochastic and correlated failures due to power or network node failure on job reliability. We also propose an approximation technique for estimating a given job's reliability.

2. A scheduling algorithm to approximate the number of tasks required to guarantee that a job will achieve the desired reliability, and to select a subset of machines to run those tasks.

3. Analytical proofs of the algorithm's validity and a simulation-based illustration and evaluation of the algorithm for a cluster with different failure sources and jobs with different target reliabilities. The evaluation shows that the algorithm can effectively approximate the minimum number of extra tasks required to guarantee a job's reliability.

2 Problem formulation

A job J arrives at a data center and executes a group of identical tasks [8]. This is a common application model for datacenters, where large numbers of data analytics jobs (e.g. MapReduce [9] or Spark [10] jobs) perform parallel computations on large datasets. Each task has an expected compute and memory demand of C and M, respectively. The job starts at time t_1 and runs for time T. A job is successful if the probability that at least K tasks will be running at all times during the $[t_1, t_1 + T]$ time interval is greater than or equal to $S_{\min}$. In addition to the required K tasks, a number of *extra tasks* can be created to increase the expected probability of success. A set of tasks may fail (and stop running) simultaneously if they share a common source of potential failure such as a common power node or a common network component.

The tasks should be deployed in a cluster of physical machine. Each machine p has an available CPU and memory capacity of c_p and m_p, respectively. Machines are connected by a set of network nodes $\bar{R}$ and the electrical power is supplied by a set of power nodes $\bar{W}$. Failure of a power node causes failure on all the associated machines and the tasks deployed on those machines.

Given a fixed power and network topology, for each job J with desired number of running tasks K, we want to determine n, the required number of extra tasks, and $\mathbf{x}$, a placement of $N = K + n$ tasks on the machines, so at least K tasks are running at all times during the job runtime, with a certain probability $S_{\min}$. The job reliability is defined as the probability of the job operating for a certain amount of time, with at least K running tasks, at all times. In our model, each machine only runs one task. Task cannot be restarted and their failures are terminal. The jobs cannot produce an output in the event of a partial failure.

Given a placement vector $\mathbf{x}$ representing the distribution of tasks over different failure domains, and $S(\mathbf{x})$ as the reliability of the job, the goal is to find $n^* = \min \{n \mid S(\mathbf{x}) \geq S_{\min}\}$ during the job's runtime T, subject to the relevant capacity constraints.

2.1 Network and power topology

We model the dependencies among system components as a 2-level multi-rooted tree, where the leaves are the machines and the parents are power and network nodes. We assume that the power and network topology may contain redundancy, where each machine can be connected to an extra network node or be supported by an extra power node. In such cases, a machine is functional as long as at least one power node and one network node are available.

We also define a network failure domain R as the set of machines that share the same network components and are thus at risk of concurrent failure. Similarly, we define a power domain W as the set of machines that share the same power nodes.

We define r_i as a random binary variable, where $r_i = 1$ if the network failure domain i is available and running, and $r_i = 0$ otherwise. The network failure domain is available as long as one network component is functional. Similarly, we define the random binary variable w_j, where $w_j = 1$ if the power failure domain j is available and $w_j = 0$ otherwise. The power failure domain is available as long as one power node is functional.

In addition, we define the failure domain F_l, as the set of tasks deployed in a power failure domain that are also connected via a single network failure domain. The binary random variable $f_l = 1$ if at least one network node and one power node in F_l are operational. Figure 1 show a data center topology with respect to network and power failure domains, showing the redundancies in the system.

The network and power nodes break independently and are not repairable. The time to failure of each network node during a job runtime T is exponentially distributed with a random failure rate of $0.00022 \leq \lambda \leq 0.00032$ per hour. Similarly, the time to failure of the power nodes (i.e. the PDUs) during a job with runtime T is assumed to be exponentially distributed over time with a failure rate of $\lambda = 0.4 \times 10^{-6}$ per

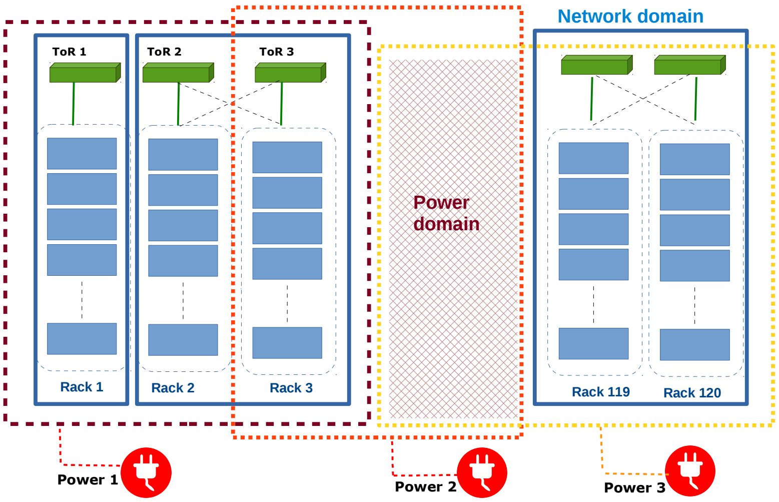


Figure 1: Data center topology with respect to different failure domains.

hour [11]. We also assume that all machines have an identical probability of hardware failure.

2.2 The reliability model

To model the impact of correlated failures on reliability, we specify the probability of each subset of tasks being unavailable. Let us assume that x_l is the number of tasks running in a failure domain F_l, $\mathbf{x} = [x_1, x_2, \ldots, x_L]$ is the placement vector of the tasks over the failure domains. The quantity x_l also happens to be the number of tasks that fail when failure domain F_l is not working. Moreover, $\mathbf{f} = [f_1, f_2, \ldots, f_L]$ is a failure state vector showing the availability or failure of the failure domains and L is number of failure domains. The number of running tasks can then be calculated as:

$$N(\mathbf{f}, \mathbf{x}) = \sum_{l=1}^{L} f_l x_l \tag{1}$$

Let

$$P^R(\mathbf{r}) = \prod_{\{i|r_i=1\}} A_i^R(T) \prod_{\{i|r_i=0\}} (1 - A_i^R(T)) \tag{2}$$

$$P^W(\mathbf{w}) = \prod_{\{j|w_j=1\}} A_j^W(T) \prod_{\{j|w_j=0\}} (1 - A_j^W(T)) \tag{3}$$

where $P^R(\mathbf{r})$ and $P^W(\mathbf{w})$ are the probabilities of network failure domains and power failure domains for failure state vectors $\mathbf{r} = [r_1, r_2, \ldots, r_R]$ and $\mathbf{w} = [w_1, w_2, \ldots, w_W]$, respectively. In addition, $A_i^R(T)$ and $A_j^W(T)$ are the reliabilities of network failure domain i and power failure domain j for a job with T runtime, respectively. The reliability of the job, $S(\mathbf{x}) \geq P(N(\mathbf{f},\mathbf{x}) \geq K)$, can be calculated as the sum of the probabilities of all possible combinations of failure events when the number of running tasks $N(\mathbf{f},\mathbf{x}) \geq K$. This can be written as:

$$S(\mathbf{x}) = \sum_{\{(\mathbf{r},\mathbf{w})|N(\mathbf{f},\mathbf{x})\geq K\}} P^R(\mathbf{r})P^W(\mathbf{w}) \tag{4}$$

In this formulation, the component failures are independent of each other. However, each component failure can terminate all tasks in one or more failure domains.

2.3 Approximating the reliability value

In practice, computing the exact reliability value is complex and computationally expensive. Therefore, we propose a method for approximating the reliability value.

The reliability function $S(\mathbf{x})$ is a step function; it is assumed that a job's reliability increases in discrete steps as the number of extra tasks is increased. However, different types and combinations of failures do not necessarily reduces the reliability value to the same extent. A network node has a much greater failure probability than a power node, but fewer extra tasks are required to mitigate the impact of a network node failure: network component failures disconnect 60 to 120 machines (one or two racks) whereas a Power Distribution Unit (PDU) failure leads to the outage of 20 to 60 racks [12].

We reformulate equation (4) as a sum over the total number of failures in the system. To do this, for $i = 0,1,\ldots,R$ and $j = 0,1,\ldots,W$, we define:

$$S_{i,j}(\mathbf{x}) = \sum_{\left\{(\mathbf{r},\mathbf{w})| \begin{array}{c} \|\mathbf{r}\|_1 = R-i \\ \|\mathbf{w}\|_1 = W-j \\ N(\mathbf{f},\mathbf{x}) \geq K \end{array}\right\}} P^R(\mathbf{r})P^W(\mathbf{w}) \tag{5}$$

In this equation, $S_{0,0}(\mathbf{x})$ is the reliability of the job if none of the network or power failure domains fail, and $S_{i,j}(\mathbf{x})$, $1 \leq i \leq R$ and $1 \leq j \leq W$ is the probability that the job has $N(\mathbf{f},\mathbf{x}) \geq K$ tasks, given i network failures and j power failures. By combining definitions (4) and (5), we get the following expression for the reliability

$$S(\mathbf{x}) = \sum_{i=0}^{R}\sum_{j=0}^{W} S_{i,j}(\mathbf{x}) \tag{6}$$

The probability of failure of multiple components, $S_{i,j}(\mathbf{x})$, depends on the failure probabilities of the individual components, each of which is small. Considering the probability of failures of the components in our model, we can argue:

$$S_{0,0}(\mathbf{x}) \gg S_{1,0}(\mathbf{x}) \gg S_{2,0}(\mathbf{x}) \gg \ldots S_{0,1}(\mathbf{x}) \ldots \gg S_{R,W}(\mathbf{x}) \tag{7}$$

Therefore, as i and j increase, the improvement in system reliability, $S_{i,j}(\mathbf{x})$, becomes progressively smaller. There is thus an optimal number of extra tasks; adding further tasks above this threshold would not contribute substantially to the system's reliability. In other words, as the probability of a high number of failures during a job's lifetime decreases, so too does the need to plan and assign extra tasks to achieve reliability.

The fact that the reliability function increases in discrete steps is useful when approximating the reliability value and estimating the number of extra tasks required to achieve a given reliability. Each step in the reliability function corresponds to a failure arrangement, $(\mathbf{r}, \mathbf{w})$, which specifies the type and number of failures that the existing arrangement is sufficient to cover. A desired reliability can then be achieved by providing the minimum redundancy required to cover the corresponding failure arrangement.

The approximation becomes essential, as the computational expense of calculating $S(\mathbf{x})$, which necessitates calculation of every possible combination of failures that satisfy $N(\mathbf{f}, \mathbf{x}) \geq K$. To reduce the computational complexity, we approximate the sum on the right hand side of equation (6) by discarding the terms $S_{i,j}(x)$ that correspond to failures of components (i, j) whose probability of failure is negligible with respect to S_{min}. This enables us to obtain approximate reliability values without performing expensive computations, and also helps us estimate the number of extra tasks required to prevent failure events that have non-negligible effects on reliability relative to S_{min}.

3 Fault-aware Scheduling

To schedule a job in a way that achieves a given reliability, we must approximate the minimum necessary number of extra tasks and identify a placement that satisfies the reliability constraint $S(\mathbf{x}) \geq S_{min}$. Our approximation algorithm is based on the discussion in the previous section. We aim to determine which failures of (i, j) must be compensated for to guarantee the target reliability S_{min}. Having identified this set of essential failures, we must then provide the minimum level of redundancy necessary to cover them.

To identify the failures for which it is necessary to provide redundancy, we have implemented a decision tree. Each node in the tree corresponds to a failure of (i, j) components, i.e. a specific number of failures of given types. For each node of the tree, we estimate the number of extra tasks required to compensate for the corresponding failures, n, and approximate the job reliability for the current arrangement. The algorithm starts at the root of the tree, (0,0), and initially expands along the power branch,(0,1). This is done because providing redundancy for power outages with appropriate task placement also automatically protects against the negative impacts of certain multiple network failures. If the target reliability, S_{min}, cannot be achieved by covering for one power failure, no amount of additional redundancy with respect to network failure would be sufficient to compensate for this deficiency, so it is necessary to expand further along the power branch. However, if providing redundancy for one power domain failure results in $S(\mathbf{x}) > S_{min}$, there is a chance of obtaining the desired reliability with fewer extra tasks by covering for some network domain failures. If this is the case, we expand along the network branch and iteratively increase the number of possible component failures, (i, j), adding the necessary redundancy to cover these failures at each step. In each expansion, we re-compute the new $\mathbf{x}$ for new $N = K + n$ and its associated $S(\mathbf{x})$. We stop at the node that satisfies the target reliability S_{min}. Note that, $\mathbf{x}$ is the placement vector representing the distribution of tasks over the type of failure that we try to cover. If the required redundancy is to support network failures, then $\mathbf{x}$ is the distribution of tasks over the network failure domains. However, if the required redundancy is to support power failures, then $\mathbf{x}$ is the distribution of tasks over the power failure domains.

The n values of the last two traversed nodes are the lower and upper bound estimates of the approximate minimum number of extra tasks required to achieve $S(\mathbf{x}) \geq S_{min}$. Having determined this interval, we can easily approximate the minimum by performing a bisection search over the n values that satisfy $S(\mathbf{x}) \geq S_{min}$. The number of iterations is limited and small because the intervals are usually limited and small. This algorithm is outlined graphically in Figure 2 and more precisely in Algorithm 1.

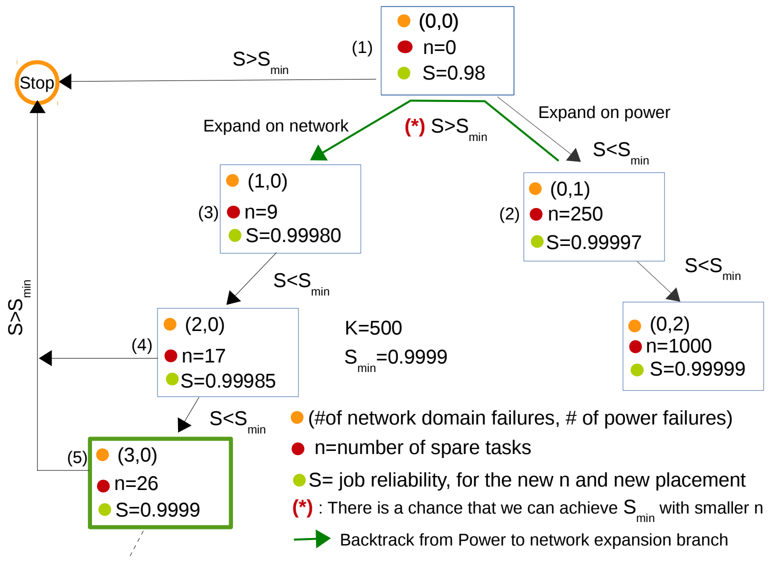


Figure 2: Decision tree for identifying the required redundancy level.

Algorithm 1 Fault-aware scheduling algorithm

```
 1: procedure SCHEDULING
 2:     Calculate $S_{0,1}(\mathbf{x})$ assuming $n = \frac{K}{W-1}$, i.e. the reliability of a job in the event of one
    power failure
 3:     if $(S > S_{min})$ then
 4:         Branching($S_{1,0}(\mathbf{x})$); Expand along the network branch
 5:     else $S < S_{min}$
 6:         Branching($S_{0,2}(\mathbf{x})$); Expand along the power branch
 7:     end if
 8: end procedure
 9:
10: procedure BRANCHING($S_{LevelNetwork,LevelPower}(\mathbf{x})$)
11:     if $S > S_{min}$ then return x
12:     end if
13:     if LevelNetwork > 0 then
14:         calculate $n = \frac{K \times LevelNetwork}{R - LevelNetwork}$
15:         x := Schedule $Rn$ tasks equally on $R$ network failure domains, considering the
    capacity constraints
16:         Sort network failure domains according to their reliability
17:         Update x after removing $N^* - N$ tasks from the least reliable domains
18:         calculate job reliability $S$
19:     else
20:         if LevelPower > 0 then
21:             calc $n = \frac{K \times LevelPower}{W - LevelPower}$
22:             Schedule $Wn$ tasks on $W$ power failure domains
23:             Sort power failure domains according to their reliability
24:             remove $N^* - N$ from the least reliable domains
25:             calculate job reliability $S$
26:         end if
27:     end if
28:     if $S_{LevelNetwork,\, LevelPower}(\mathbf{x}) < S_{min}$ and (LevelNetwork > 0) then
29:         Branching($S_{LevelNetwork++,\, LevelPower}(\mathbf{x})$)
30:     else
31:         if $S_{(LevelNetwork,\, LevelPower)}(\mathbf{x}) < S_{min}$ and (LevelPower > 0) then
32:             Branching($S_{LevelNetwork,\, LevelPower++}(\mathbf{x})$)
33:         end if
34:     end if
35: end procedure
```

3.1 Estimating the number of extra tasks

To estimate the number of extra tasks n, let us first assume that the desired reliability S_{min} can be guaranteed by covering a single network failure domain:

Lemma 1. *To provide full redundancy for one network domain failure, the required number of extra tasks is* $n = \lceil \frac{K}{R-1} \rceil$.

Proof. Let N be the total number of deployed tasks for the job. To ensure that at least K tasks are running if any single network failure domain fails, at least K tasks must run on each collection of $R-1$ network failure domains. By the pigeon hole principle [13], at least one of these $R-1$ network failure domains has $n = \lceil \frac{K}{R-1} \rceil$ tasks. To ensure redundancy, the remaining $R-1$ network failure domains must have at least K tasks, so we need $K+n$ tasks to ensure redundancy.

Furthermore, we need to prove that this setting is realizable. In other words, we should be able to place $K+n$ tasks over the R network failure domains such that no domain has more than n tasks. One straightforward way of doing this is to put n tasks on each network failure domain, resulting in $R\lceil \frac{K}{R-1} \rceil \geq K+n$ tasks in total, and then remove any $(Rn)-K-n$ tasks. $\square$

Lemma 2. *To provide full redundancy for* $i > 0$ *network domain failures, the number of required extra tasks is* $n_r = \lceil \frac{Ki}{R-i} \rceil$.

Proof. Lemma 2 is an extension of the statement in Lemma 1. For a job to survive $i > 0$ network domain failures, we need at least K tasks running on $R-i$ network failure domains. Assuming that each i network failure domain forms a larger network failure domain, we can argue that at least one of these larger domains has $n = \lceil \frac{Ki}{R-i} \rceil$ tasks. In other words, there is a set of i domains that has at least $\lceil \frac{Ki}{R-i} \rceil$ tasks running on them.

$\square$

Using a similar argument, we can conclude that the number of extra tasks required to provide full redundancy in the event of $j > 0$ power failures is $n_w = \left\lceil \frac{Kj}{W-j} \right\rceil$. The only difference is that deploying $n_w \gg n_r$ tasks automatically covers for the failure of R^* network failure domains due to the deployment of a greater number of spare tasks. Using the equation in Lemma 2, we can find R^*, the number of network domain failures covered by deploying n_w extra tasks:

$$n_w = \left\lceil \frac{KR^*}{R-R^*} \right\rceil \rightarrow R^* = \left\lceil \frac{n_w}{K+n_w} \times R \right\rceil \tag{8}$$

Therefore, when estimating the increase in reliability it is necessary to consider the coverage of both the power failures and R^* network failure domain failures.

3.2 Die-Hard Placement

To identify a suitable placement given the failure of (i, j) components, we must ensure that at least K jobs are running on each combination of failure domains. One intuitive approach is to evenly assign n tasks to each domain. Assuming that placement is done over network failure domains, this results in at most $Rn \geq K + n$ tasks in total given the capacity constraints. Next, we can remove $(Rn) - K - n$ tasks from the assigned tasks. Since n is derived independently from the failure probabilities of each failure domain, a lower n can be obtained by removing the tasks from the failure domains with highest probability of failure. We continue reducing the number of tasks from the least reliable failure domains as long as the reliability condition $S(\mathbf{x}) \geq S_{min}$ holds.

If the placement is to be done over power failure domains, we similarly assigns tasks to each power failure domain. However, we further distribute the tasks to the network failure domains within the power failure domain to cover the

4 Experimental Setup

We simulated a cluster with 3 power nodes and 9600 machines connected through 120 Top of Rack (ToR) switches. Each machines has 4 CPU cores and 16 GB of memory, and a background load that is a random value uniformly chosen within the available capacity range. Details of the power and network topology are presented in Section (2.1). On the basis of the described topology, the cluster's machines are organized into 61 network failure domains and 4 power failure domains.

A job arrives at the system and runs for T hours. The value for the T is introduced in the experiments. All tasks have identical CPU and memory demands, uniformly chosen from 1 to 4 cores for the CPU and 1 to 16 GB for memory. The target reliability for the job was set to 0.999 or 0.9999, depending on the experiment.

The time to failure of each network node during a job runtime T is exponentially distributed with a random failure rate of $0.00022 \leq \lambda \leq 0.00032$ per hour. Similarly, the time to failure of the power nodes (i.e. the PDUs) during a job with runtime T is assumed to be exponentially distributed over time with a failure rate of $\lambda = 0.4 \times 10^{-6}$ [11] per hour. The results presented below are average values obtained from 10 separate runs of each experiment.

5 Illustration and evaluation

We have already presented an analytical proof of the validity of our approach for computing the number of required extra tasks. This section therefore illustrates the impact of replica count on job reliability, for a typical use case. We also study the impact of different placement strategies on job reliability and the required number of extra tasks.

5.1 Impact of the number of replicas on reliability

As shown in Equation (7), it is not necessary to provide full redundancy for a job to meet the required reliability level. By exploiting this property of the reliability model, it is possible to reduce the complexity of the computations and facilitate the estimation of the required number of extra tasks. To illustrate this point, we studied the impact of increasing the number of extra tasks on reliability. As shown in Table 1, as n increases the improvement in job reliability becomes progressively smaller and ultimately negligible. The results also show that there is an optimal number of replicas and further increasing the number of replicas does not significantly increase job reliability. The optimal number increases in a stepwise fashion and is related to the number of failures that can be tolerated.

Table 1: *Impact of number of extra tasks on job reliability, K=1000*

No. of replicas (n)	Reliability	Average improvement/replica
0	0.9619771697	
16	0.9884874963	1.66×10^{-3}
33	0.9997473744	6.62×10^{-4}
51	0.9999373772	1.05×10^{-5}
70	0.9999376013	1.17×10^{-8}
89	0.9999376019	3.15×10^{-11}

5.2 Impact of the scheduling strategy

Given a total number of tasks $N = K + n$, there are a number of distributions that can satisfy the reliability constraint $S(\mathbf{x}) \geq S_{min}$. Any distribution is acceptable as long as it guarantees a total of K tasks running on all possible combinations of available failure domains. To find a suitable distribution we have introduced a placement algorithm, *Die-hard (DH)*, which is described in Section 3.2. We compare *DH* algorithm to two other intuitive placement strategies. We observe that, for high reliability targets, some quite intuitive placement strategies do not necessarily satisfy the reliability constraints with the approximated number of replicas.

The three placement strategies are:

- **DH**: The algorithm initially assigns an equal number of tasks over different failure domains. It then iteratively removes tasks from the domains with the highest probabilities of failure provided that the reliability constraints hold.

- **Proportional placement**: The algorithm distributes the tasks among failure domains in proportion to their probabilities of failure.
- **Highest reliability first (HRF)**: the *HRF* algorithm ranks the domains based on their failure probability. It then places the tasks on the domains with the greatest reliability provided that they have available capacity.

To compare the three algorithms, we use the *affinity score* [2], as a metric to measure the likelihood of correlated failures. Let $\mathbf{x} = [x_1, ..., x_l]$ be the distribution of tasks over different failure domains, where $x_1 \leq x_2 \leq ... \leq x_l$. Let the impact of the correlated failure be the number of tasks sharing a common failure source. The affinity score is:

$$\sum_{i=1}^{l} \frac{x_i(x_i - 1)}{2} \tag{9}$$

The affinity score is maximized when all the tasks (task failures) are in the same domain, and minimized when tasks are spread over different domains. A low affinity score represents a low concentration and a low risk of correlated failure.

Table 2 compares the required numbers of extra tasks, reliabilities, and affinity scores of the three placement algorithms. For high reliability targets, the *DH* algorithm clearly requires the least extra replicas to guarantee reliability. The *DH* algorithm gives the lowest affinity score, showing that it reduces the risk of correlated failures by having the lowest task concentration. Reducing task concentration increases reliability with respect to correlated failures while minimizing the number of extra tasks required to guarantee the desired reliability.

It can also be seen that, for the same reliability target (0.9999), the *HRF* algorithm has the lowest reliability and the highest affinity score. Although it may seem intuitive to place as many tasks as possible on the domain with the highest reliability, this placement strategy leads to the highest affinity score and the lowest reliability value when there is a risk of correlated failure. It can also be seen that, for high reliability targets, the *HRF* algorithm cannot even satisfy the reliability constraint with the same upper bound value n as the other two placement strategies. Thus, placing tasks using the *HRF* algorithm would substantially increase the number of extra tasks required for each job to achieve a given level of reliability. This is because the *HRF* placements yield a high level of correlation among potential failures, reducing the benefits of increasing redundancy. Replication alone is thus not sufficient to protect against correlated failures. In other words, the risk of correlated failure is not mitigated and reliability is not improved by increasing the redundancy within the failure domain.

However, as shown in Table 3, if job's desired reliability is lower than the reliability of a failure domain, (in this case 0.999), deploying all the tasks on any failure domain with higher reliability that the job's target reliability satisfies the $S(\mathbf{x}) \geq S_{min}$ constraint with no extra tasks. It should also be noted that the probability of failure of a component during a job runtime is a function of the job's duration. In other words, the longer the job, the more probable to have more failures during its runtime. Therefore, for long running jobs with high reliability targets, distributing the tasks over different failure domains, (using *DH* or *proportional* placement), is necessary to satisfy the job's reliability. However, this may not be the case for a short-running job, as it is more probable to satisfy the reliability constraint by deploying the tasks using *HRF*. The impact of job duration on required number of extra tasks and job reliability is shown in Table 4.

Table 2: *Impact of placement on required number of extra tasks, reliability and affinity score,* $S_{min} = 0.9999$*, job duration = 80 hours.*

K	Algorithm	Extra tasks (n)	Affinity score	Reliability
500	DH	18	2089	0.9999
	Proportional	23	2274	0.9999
	HRF	25	137026	**0.9995**
700	DH	32	4015	0.9999
	Proportional	33	4476	0.9999
	HRF	36	269745	**0.9996**
1000	DH	36	8280	0.9999
	Proportional	45	9228.6	0.9999
	HRF	51	550725	**0.9996**

Table 3: *Impact of placement on required number of extra tasks, reliability and affinity score,* $S_{min} = 0.999$*, K = 500, job duration = 80 hours.*

Algorithm	Extra tasks (n)	Affinity score	Reliability
DH	9	2022	0.9995
Proportional	12	2170	0.9995
HRF	0	124251	0.9995

It can also be seen that the *proportional* placement algorithm requires more extra tasks than the *DH* algorithm. This is because although it distributes tasks proportionally over different domains, the distribution is still biased by the domains' reliability. This bias in distribution increases the concentration of the placement scheme and thus the affinity score, thereby reducing reliability. The decrease in reliability forces the system to deploy more tasks to achieve the target reliability. In other words, a minimum level of distribution is required to achieve a high reliability.

6 Related work

Below, we review two categories of related work:

Table 4: *Impact of job duration and placement on required number of extra tasks, reliability and affinity score, $S_{min} = 0.999$, K = 500.*

Duration	Algorithm	Extra tasks (n)	Affinity score	Reliability
80 h	DH	9	2022	0.9995
	Proportional	12	2170.6	0.9995
	HRF	0	124251	0.9995
168 h	DH	18	2058	0.9995
	Proportional	21	2247	0.9992
	HRF	25	137026	**0.998**

6.1 Failure analyses in data centers

Several studies [2], [14], [5] have focused on characterizing failure sources and analyzing their impact on system availability and reliability in cloud data centers. Ford et al. [2] studied the impact of correlated failures on availability for Google's cloud storage system. They argued that scheduling strategies should be aware of failure bursts caused by correlated failures. Assuming that machines fail independently results in over-estimation of the system's availability at least by two orders of magnitude. They also developed an availability model using Markov chains and introduced multi-cell replication schemes to cope with correlated failures.

Similarly, Gill et al. [14] presented an analysis of possible failures in a Microsoft cloud data center. They also studied the effectiveness of redundancy at maintaining reliability. Their observations indicated that the effectiveness of network redundancy at masking network failure is only 40%. This was attributed to the propagation of configuration errors, which can lead to concurrent failures of many tasks. Their results highlight the necessity of spreading tasks over different domains of control to achieve high reliability.

6.2 Failure-aware scheduling and allocation

Cirne et al. [15] discussed a task backup strategy to provide reliability guarantees for a job. The goal is to determine the probability of losing a certain number of backups and use this probability for the admission control decisions. They took into account the possibility of correlated failures of tasks caused by rack and machine faults. However, they supported the correlated failures caused by the rack failures as long as the failure domains nicely nest into a tree. They also did not fully investigate the possibility of multiple rack failures during the job runtime and its impact on job reliability.

Bakkaloglu et al. [16] studied correlated failures in storage systems. They modeled availability using a beta-binomial distribution, which was computed by randomizing the failure probabilities according to a binomial distribution, and used a correlation factor to quantify the intensity of correlations. However, different studies [2, 4] have shown that beta-binomial distributions do not provide a good fit to real-world data from data centers. Moreover, it is still challenging to accurately estimate correlation coefficients.

Tang et al. [3] analyzed the impact of correlated failures on reliability for DEC VAX clusters, and found that such failures can reduce reliability by several orders of magnitude. They proposed a correlation coefficient-based model to quantify the relationship between failures and reliability. However, the proposed model is only applicable to two-way correlations and is not straightforwardly generalized to higher levels of correlation.

Bodik et al. [7] presented an optimization framework for achieving high fault tolerance while reducing the bandwidth consumption in the network. They improved fault tolerance by spreading applications across different failure domains. However, their framework is not designed to cope with the problem of correlated failures and does not take probabilities of failure into account during scheduling.

Rabbani et al. [17] proposed a management framework for maintaining high reliability. Their method considers the heterogeneity of components' failure rates when planning the number and allocation of redundant virtual nodes in a virtual infrastructure. However, their main focus is on independent failures and they have not considered the impact of correlated failures on reliability. Moreover, their main objective is to minimize the number of machines required to deploy the service, at the expense of increasing the number of backups. Their procedure iteratively increases the number of required backups, which is assumed to be the minimum number of vitual machines deployed in machines in each round, until the reliability constraint is satisfied. We believe that estimating the number of extra backups without considering allocations and the probability of machine failures leads to over provisioning and is not a reasonable way to maintain reliability.

Venice [18] is a framework for achieving high reliability for a 3-tier application with VM dependencies. It has a reliability-aware scheduler that deploys VMs on the machines with the lowest reliability capable of meeting the service reliability requirement. Then, over a number of trials, it removes the machines with the lowest reliability and deploys the VMs on the remaining set of machines. Finally, the scheduler selects the allocation scheme with the lowest cost as its final solution. Sampaio [19] also considered the Mean Time Between Failures (MTBF) of the nodes when planning allocations. Both of these works ignored the impact of correlated failures on service reliability and also did not consider the benefits of using redundant replicas to attain the required service reliability.

Mills et al. [20] addressed the replica scheduling problem using a greedy heuristic in a tree structure. The tree structure represents the dependencies among system components. The aim is to minimize the number of concurrent component failures due to a single failure event. However, structuring the component dependencies as a tree represents a considerable simplification because it assumes that either there is a single source of failure or that the power and network topologies map onto one another. In other words, the tree structure ignores the probability of nested and overlapping failure domains.

7 Conclusion

In this paper we address the problem of efficiently scheduling resources in a cloud data center to achieve reliability guarantees even in the face of correlated failures. The goal is to guarantee each job's reliability while minimizing the number of extra tasks required during the job's runtime. The reliability guarantee is achieved through task replication and diversified job placement over different failure domains.

We present a reliability model that accounts for failure probabilities and the topologies of power and network components in the data center. We also provide a method for obtaining approximate reliability estimates that does not require expensive computations.

We use our model to approximate the minimum number of extra tasks required to guarantee a desired reliability. This is done by using a decision tree to map the target reliability to a specific redundancy level. Moreover, we introduce a scheduling algorithm to schedule tasks on resources in a way that accounts for their capacity constraints. Using analytical proofs and simulations, we show that the algorithm can effectively approximate the minimum number of extra tasks required to guarantee job reliability.

Acknowledgement

Financial support for this work was provided in part by the Swedish Research Council (VR) under contract number C0590801 for the project Cloud Control, the Swedish Government's strategic research project eSSENCE, and the European Union's Seventh Framework Programme under grant agreement 610711 (CACTOS). We also want to thank Alessandro Papadopoulos for suggesting an interesting title.

References

[1] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "Vl2: a scalable and flexible data center network," in *ACM SIGCOMM Computer Communication Review*, vol. 39, pp. 51–62, ACM, 2009.

[2] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan, "Availability in globally distributed storage systems.," in *Operating Systems Design and Implementation, (OSDI)*, pp. 61–74, 2010.

[3] D. Tang and R. K. Iyer, "Analysis and modeling of correlated failures in multicomputer systems," *IEEE Transactions on Computers*, vol. 41, no. 5, pp. 567–577, 1992.

[4] P. Yalagandula, S. Nath, H. Yu, P. B. Gibbons, and S. Seshan, "Beyond availability: Towards a deeper understanding of machine failure characteristics in large distributed systems," in *Proceedings of the Workshop on Real, Large Distributed Systems (WORLDS '04)*, 2004.

[5] B. Chun and A. Vahdat, "Workload and failure characterization on a large-scale federated testbed," *Intel Research Berkeley Technical Report IRB-TR-03-040*, 2003.

[6] B. Schroeder, E. Pinheiro, and W.-D. Weber, "Dram errors in the wild: a large-scale field study," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 37, pp. 193–204, ACM, 2009.

[7] P. Bodík, I. Menache, M. Chowdhury, P. Mani, D. A. Maltz, and I. Stoica, "Surviving failures in bandwidth-constrained datacenters," in *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, pp. 431–442, ACM, 2012.

[8] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg," in *Proceedings of the Tenth European Conference on Computer Systems*, pp. 18–35, ACM, 2015.

[9] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[10] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, vol. 10, p. 10, 2010.

[11] H. Geng, *Data Center Handbook.* Wiley.

[12] X. Fan, W.-D. Weber, and L. A. Barroso, "Power provisioning for a warehouse-sized computer," in *ACM SIGARCH Computer Architecture News*, vol. 35, pp. 13–23, ACM, 2007.

[13] W. A. Trybulec, "Pigeon hole principle," *Journal of Formalized Mathematics*, vol. 2, no. 199, 1990.

[14] P. Gill, N. Jain, and N. Nagappan, "Understanding network failures in data centers: measurement, analysis, and implications," in *ACM SIGCOMM Computer Communication Review*, vol. 41, pp. 350–361, ACM, 2011.

[15] W. Cirne and E. Frachtenberg, "Web-scale job scheduling," in *Job Scheduling Strategies for Parallel Processing*, pp. 1–15, Springer, 2013.

[16] M. Bakkaloglu, J. Wylie, C. Wang, and G. Ganger, "On correlated failures in survivable storage systems (2002). Technical report Carnegie Mellon University, cmu-cs-02-129."

[17] M. G. Rabbani, M. F. Zhani, and R. Boutaba, "On achieving high survivability in virtualized data centers," *IEICE Transactions on Communications*, vol. 97, no. 1, pp. 10–18, 2014.

[18] Q. Zhang, M. F. Zhani, M. Jabri, and R. Boutaba, "Venice: Reliable virtual data center embedding in clouds," in *Proceedings IEEE INFOCOM 2014*, pp. 289–297, IEEE, 2014.

[19] A. M. Sampaio and J. G. Barbosa, "Towards high-available and energy-efficient virtual computing environments in the cloud," *Future Generation Computer Systems*, vol. 40, pp. 30–43, 2014.

[20] K. A. Mills, R. Chandrasekaran, and N. Mittal, "Algorithms for replica placement in high-availability storage," *arXiv preprint arXiv:1503.02654*, 2015.