TECHNISCHE UNIVERSITÄT MÜNCHEN
Institut für Informatik
Lehrstuhl für Informatik XVIII

Energy efficient capacity management in virtualized data centers

Andreas Wolke

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzende:		Univ.-Prof. Dr. Claudia Eckert
Prüfer der Dissertation:		
	1.	Univ.-Prof. Dr. Martin Bichler
	2.	Univ.-Prof. Dr. Georg Carle

Die Dissertation wurde am 22.09.2014 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 16.01.2015 angenommen.

Abstract

With a growing demand for computing power and at the same time rising energy prices, the cost-efficient operation of virtualized data centers becomes increasingly important. In the intention to reduce operating costs, data center operators are trying to minimize the total active server demand. In this work we evaluated the performance of a variety of VM allocation controllers in two commonly found scenarios. For enterprise data centers we could show that static allocations have advantages over dynamic ones. Static allocations work well for predictable workloads, which are typically found in enterprise environments with a daily or weekly workload seasonality. Dynamic approaches deliver good results for unpredictable workloads but need to trigger VM migrations, which can decrease service quality because they entail resource demands on their own. On-demand cloud scenarios with an unpredictable stream of incoming and outgoing VM allocation requests benefit by dynamic strategies, however. We found that dense packing followed by subsequent migrations in case of overload to be an effective capacity management strategy. Our analysis outcomes are based on an extensive set of simulations and experiments that were conducted in a testbed infrastructure.

Zusammenfassung

Mit zunehmender Nachfrage an Rechenleistung bei zugleich steigenden Energiepreisen gewinnt der kosteneffiziente Betrieb virtualisierter Rechenzentren an Bedeutung. Zur Senkung von Investitions- und Energiekosten soll der Serverbedarf mithilfe intelligenter Strategien zur Allokation virtueller Maschinen (VMs) minimiert werden. Diese Arbeit beleuchtet die Effizienz verschiedener VM-Allokationsstrategien anhand zweier gängiger Szenarien. Für betriebliche Rechenzentren konnte gezeigt werden, dass statische Strategien den dynamischen vorzuziehen sind. Sie nutzen bekannte Muster aus Lastaufzeichnungen um eine langfristige Allokation virtueller Maschinen zu berechnen. Dynamische Ansätze migrieren VMs zwischen Servern und erzielen somit besonders bei unbekannten Lastmustern gute Ergebnisse. Die anfallenden Migrationen benötigen ihrerseits jedoch zusätzliche Ressourcen mit negativen Auswirkungen auf das Gesamtsystem. Ein anderes Bild ergibt sich für On-Demand-Szenarien mit einem variablen Strom an eingehenden und abgehenden VM-Allokationsanfragen. Eintreffende VMs sollten initial möglichst dicht auf die Server verteilt werden. Komplementär führen dynamische Ansätze einen Lastausgleich durch und gleichen Unzulänglichkeiten der initialen Platzierung aus. Die Erkenntnisse beider Szenarien stützen sich auf Simulationen sowie umfangreiche experimentelle Studien die im Rahmen dieser Arbeit durchgeführt wurden.

Published work

Most sections of this thesis have been developed with co-authors and have already been published in papers. In all cases, my contribution to the work is significant. The individual sections of this thesis are based on the following list of publications.

- Chapter 2 – A. Wolke, M. Bichler, and T. Setzer, “Planning vs. dynamic control: Resource allocation in corporate clouds,” in IEEE Transactions on Cloud Computing (TCC), Accepted, 2014. [98]
- Chapter 3 – A. Wolke and L. Ziegler, “Evaluating dynamic resource allocation strategies in virtualized data centers,” in Cloud Computing (CLOUD), 2014 IEEE 7th International Conference on, 2014. [97]
- Chapter 4 – T. Setzer and A. Wolke, “Virtual machine re-assignment considering migration overhead,” in 2012 IEEE Network Operations and Management Symposium, 2012, pp. 631–634. [78]
- Chapter 5 – A. Wolke and C. Pfeiffer, “Improving enterprise VM consolidation with high-dimensional load profiles,” in Cloud Engineering (IC2E), 2014 IEEE International Conference on, 2014. [95]
- Chapter 6 – M. Seibold, A. Wolke, M. Albutiu, M. Bichler, A. Kemper, and T. Setzer, “Efficient Deployment of Main-Memory DBMS in Virtualized Data Centers,” in Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on, 2012, pp. 311–318. [74]

- Chapter 7 – A. Wolke, T.-A. Boldbaatar, C. Pfeiffer, and M. Bichler, "More than bin packing: A large-scale experiment on dynamic resource allocation in IaaS clouds," (Working Paper). [99]

- Chapter A – The simulation framework and hardware infrastructure specifications stem from: A. Wolke, M. Bichler, and T. Setzer, "Planning vs. dynamic control: Resource allocation in corporate clouds," in IEEE Transactions on Cloud Computing (TCC), Accepted, 2014. [98]

- Chapter B – A. Wolke and D. Srivastav, "Monitoring and Controlling Research Experiments in Cloud Testbeds," in Cloud Computing (CLOUD), 2013 IEEE 6th International Conference on, 2013, pp. 962–963. [96]

- Chapter C – The implementation was done in collaboration with D. Srivastav who worked on his interdisciplinary project report.

- Chapter D – J. Kroß and A. Wolke, "Cloudburst - simulating workload for IaaS clouds," in Cloud Computing (CLOUD), 2014 IEEE 7th International Conference on, 2014. As a short paper in Workshop on Information Technologies and Systems (WITS), Milan, Italy, 2013. [50]

- Section E.1 – A similar analysis was done by B. Speitkamp and M. Bichler [86]. The results reported in this thesis are done on an extended data set with a different interpretation. All illustrations and results were recreated.

- Section E.3 – A. Wolke, T.-A. Boldbaatar, C. Pfeiffer, and M. Bichler, "More than bin packing: A large-scale experiment on dynamic resource allocation in IaaS clouds," (Working Paper). [99]

- Chapter F – Controller parametrization was done in collaboration with B. Tsend-Ayush who worked on his Master's thesis.

Contents

List of Figures

List of Tables

Glossary

ACF auto correlation function.

allocation Describes the assignment of all running VMs to servers at a given time. An allocation might change continuously over time.

allocation controller An algorithm that allocates VMs to servers. Allocation controllers are categorized into static, dynamic, and placement controllers.

allocation quality A good allocation quality is considered to achieve a high allocation density while maintain a desired service quality. In our studies we assume a service quality of at least 99 %.

API application programming interface.

AR autoregressive.

CDF cumulative density function.

CMDB configuration management database.

consolidation In this thesis, consolidation describes the process of transforming physical servers to VMs. Multiple VMs are allocated on a single server in order to decrease server demand and increase server utilization.

controller An algorithm that allocates VMs to servers.

CPU central processing unit.

CSP communicating sequential processes.

DBMS database management system.

DES double exponential smoothing.

DoE design of experiments.

DRS Distributed Resource Scheduler is a VM resource allocation controller implemented by VMware to allocate resources of a compute cluster to VMs based on predefined rules and measured resource utilizations.

DSAP dynamic server allocation program.

DSAP+ dynamic server allocation program plus (+) migration overheads.

DVS dynamic voltage scaling.

dynamic allocation An allocation of VMs to servers that is under constant change. For instance, it gets changed as utilization levels of VMs change. It also describes a sequence of static allocations, each of a particular lifetime that were calculated in advance. Each subsequent allocation gets realized over time by VM migrations.

dynamic controller An algorithm that continuously optimizes the allocation of VMs to servers based on their utilization levels. It triggers VM migrations to move them between servers.

ERP enterprise resource planning.

experiment Leverages a physical server infrastructure to evaluate the performance of allocation controller implementations under realistic circumstances.

FIFO first in first out.

FSM finite state machine.

HDFS Hadoop distributed file system.

hypervisor A software that is able to operate VMs. Examples are VMware ESX [60], Xen [5], or Linux KVM.

IaaS Infrastructure-as-a-Service.

KVM kernel-based virtual machine.

LXC Linux Container.

migration Refers to the live-migration technology that moves VMs from one server to another without noticeable downtime or service disruption of the VM under migration. The technology is provided by all major hypervisors.

MRR mean reciprocal rank.

OLAP online analytical processing.

OLTP online transaction processing.

PID process identifier.

placement controller In on-demand scenarios, a placement controller determines were to create the VM for each incoming VM allocation request.

PSU power supply unit.

RAM random access memory.

simulation Programs that leverage models to resemble a physical environment such as a server infrastructure. We leverage simulations to evaluate the performance of allocation controller implementations.

SLA service level agreement.

SLO service level objective.

SPECjEnterprise2010 SPECjEnterprise2010 is a Java Enterprise application with a workload driver that is designed to benchmark the performance of Java application containers.

SSAP static server allocation program.

SSAPv static server allocation program with variable utilization.

static allocation An allocation of VMs to servers that does not change over an extended period of time (weeks, months, or years).

static controller An algorithm that gets a set of VMs and servers as input and creates a static allocation of VMs to servers.

SVD singular value decomposition.

testbed An infrastructure of multiple physical servers and network equipment that enables us to evaluate the performance of VM allocation controller implementations under real circumstances.

TPC-C A database benchmark for on-line transactional processing (OLTP).

VM A virtual machine that is running on top of a physical server which resources are shared by a hypervisor layer.

WAD web, application, and database.

workload profile In many enterprise scenarios, server utilization is seasonal on a daily or weekly basis. A workload profile is calculated on a seasonal utilization time series of a server covering multiple weeks. It describes the average utilization over all seasons.

Chapter 1

Introduction

Data center managers worldwide ranked virtualization and server consolidation as one of their top priorities in the recent years [40, 59]. In virtualized data centers, dedicated servers are replaced by VMs that are largely independent to the underlying server hardware which can handle multiple VMs at the same time. Virtualization provides a number of benefits. VMs can be allocated and deallocated automatically within seconds, allowing on-demand data centers similar to Infrastructure-as-a-Service (IaaS) clouds. Administration and maintenance costs are reduced and flexibility is increased. Virtualization enables server consolidation where multiple VMs are running on a single server to increase overall hardware utilization, thereby reducing energy and investment costs.

Energy consumption of data centers accounts for up to 50 % or more of the total operating costs [28]. It is predicted to reach around 4.5 % of the entire energy consumption in the USA [18]. A survey of the *United States Environmental Protection Agency* showed that idle servers have an energy demand between 69 % - 97 % of a fully utilized one, even if all power management functions are enabled [67]. Active servers and cooling facilities are the main energy consumers in data centers.

With the adoption of virtualization technology, server installation base stagnated while the demand for VMs grew considerably. At the same time, server administration as well as energy costs of data centers increased [42]. Automated VM to server allocation controllers promise to manage virtualized data centers in an efficient and reliable manner.

This work focuses on one major question in the area of data center virtualization: *how to allocate VMs to servers*. Primary objectives are an overall reduced energy and server demand without decreasing service quality.

Much of the academic literature focuses on computational aspects on how to allocate VMs to servers. Allocation controller implementations are evaluated based on simulations without verification by experiments in real-world settings or testbeds which mirror those. Due to complexity, simulations use simplified models which ignore latencies, virtualization overheads, and many other interdependencies between VMs, hypervisors, server hardware, or the network.

Due to these aspects, the external validity of most simulations is questionable. It is important to evaluate allocation controllers in settings as realistic as possible. As with other areas of operations research, this evaluation is very important though challenging. The setup of a testbed infrastructure which reflects a virtualized data center environment is time-consuming and expensive. This might explain the lack of experimental research results to some degree.

For this work we set up a testbed infrastructure with several servers and an elaborate management and monitoring framework. Real-world business applications were substituted by SPECjEnterprise2010, a server-benchmark that closely resembles the resource footprint of common business applications. User demand was modeled based on a set of utilization traces gathered from two European IT service providers. It allows to analyze multiple allocation controllers repeatedly under different workloads. Our goal is to achieve external validity of the experimental results.

We differentiate between two commonly found scenarios. Enterprise scenarios are addressed in Part I and describe a setting of a fixed set of VMs and servers with deterministic workloads. VM allocations and deallocations are subject to an administration schedule. On the contrary, Part II covers IaaS cloud scenarios with a setting of unpredictable workloads, arrival and departure of

VMs. A brief description and motivation for both scenarios is provided in the following.

1.1 Allocation of persistent services

Due to data security and other concerns, today's enterprises often do not outsource their entire IT infrastructure to external providers. Instead, they set up their own private or corporate clouds to manage and provide computational resources [85]. VMs are used to host transactional business applications for accounting, marketing, supply chain management, and many other functions where once a dedicated server was used.

Due to the long time of VM usage, administrators obtain detailed knowledge about typical resource demands of applications within them. Various studies such as [86, 53] have found enterprise workloads to be seasonal and therefore predictable.

Server CPU utilization in typical enterprise data centers varies between 20 % and 50 %[6]. Consolidation can reduce server demand by increasing the allocation density of VMs and server utilization. A primary question is how to allocate VMs to servers without affecting service quality. This problem is referred to as server consolidation [86, 77] or workload concentration [51] in literature.

Static controllers generate a static allocation of VMs to servers [61, 71, 79, 86]. VMs are allocated to as few servers as possible, leveraging prior knowledge of workload profiles. Such profiles describe a seasonal VM resource utilization of an average business day. A static allocation is intended to be stable for an extended time of weeks or months.

VM migration technology allows to move VMs between servers during runtime. The technology has matured to a level where it is a viable option not only for emergency situations [60], but also for routine VM allocation tasks. Another type of cloud management tools[1] leverages dynamic controllers which

[1]VMware vSphere, OpenNebula, and Ovirt

move VMs between servers based on their actual resource demand. For that purpose, they closely monitor the cloud infrastructure to detect server over- and underloads. In both cases, VM migrations are triggered to move VMs to different servers in order to dissolve the situation.

Many IT service managers consider moving to dynamic controllers [62]. In these cases it is important to understand if, and how much can be saved in terms of energy costs. This leads to our first research question –*Should managers rely on dynamic VM allocations or rather use static allocations*? Surprisingly, there is little research guiding managers on this question.

We contribute an extensive *experimental* evaluation of static and dynamic VM allocation controllers for scenarios similar to the ones found in enterprise data centers. Experiments complement and validate simulation results to provide insights on whether dynamic or static VM allocation controllers are viable options to reduce energy consumption.

Our findings show that static controllers should be preferred over dynamic ones if workload profiles are known. In all experiments, static controllers achieved higher server savings at a service quality above their dynamic counterparts. Dynamic controllers provide excellent results if workload profiles are not known or VM utilization is unpredictable. We also found dynamic controllers to deliver good results in scenarios where a huge amount of resources is utilized on a sporadic basis, e.g. for main-memory database servers. VMs leverage spare resources and are migrated to different servers as soon as these resources are actually required by the main-memory database.

1.2 Allocation of non-persistent services

As opposed to enterprise scenarios with a fixed set of VMs, IaaS cloud scenarios deal with an unpredictable stream of VM allocation and deallocation requests. No prior knowledge exists about the applications within VMs or their expected workload. VM lifetimes vary between hours, months, or even years.

Most existing cloud management[2] tools allocate incoming VMs to servers via common bin packing heuristics. VMs remain on a server until they get deallocated again. Such placement heuristics do not consider information about the actual VM resource demand, nor do they consider future allocation and deallocation requests.

Our first research question on cloud scenarios is – *What differences exist between VM placement controllers regarding allocation density?* Placement controllers allocate incoming VMs to servers. So far, none of the cloud management tools leverages VM migrations to optimize a VM allocation successively. Therefore, a second research question arises – *Do dynamic controllers have an impact on VM allocation density?*

We contribute an extensive evaluation of a variety of VM placement controllers based on various workloads. In addition, we conducted experiments in our testbed infrastructure. For simulations and experiments, we combined placement controllers with dynamic controllers from Part I to cover both research questions. The scope and scale of the experiments is beyond what has been reported in the literature, and it provides tangible guidelines for IT service managers.

Our findings are a set of allocation controllers which perform well in a wide variety of workload environments. Dynamic controllers had a substantial positive impact on the allocation density in a cloud data center. This is not obvious, as VM migrations cause additional workload on the migration source and target servers. We achieved the highest allocation density with placement controllers starting with a dense VM allocation. In case of overload, VMs were migrated and reallocated over time by a dynamic controller. If the placement decisions were based on the actual server utilization rather than the server's capacity minus VM reservations, then allocation density and server utilization could be increased.

[2]OpenStack (http://www.openstack.org/), Eucalyptus (https://www.eucalyptus.com/)

1.3 Consolidation and energy efficiency

Our primary objective in both scenarios is to automatize the allocation of VMs to servers and to reduce overall energy consumption. We assume that minimizing the total server operating hours directly translates into energy savings. This relationship is not obvious, especially considering efforts to reduce and adapt CPU energy consumption to the actual system load by techniques like dynamic voltage scaling (DVS). Modern CPUs are built to scale their energy demand almost linearly with their actual load level. However, other devices like mechanical disks, GPUs, or chipsets comprise an almost constant energy demand.

Assume a list of VM allocation and deallocation queries $L = (q_1, ..., q_n, ..., q_\nu)$ with tuples (a_n, s_n) of arrival time a_n and number of running servers s_n. Each event affects s_n as shown in Figure 1.1. Total server operating hours OH is the area under the curve as shown by Equation 1.1. Average server demand $\overline{SD}$ is the server demand weighted by time as shown by Equation 1.2.

$$OH = \sum_{n=1}^{\nu-1} s_n(a_{n+1} - a_n) \tag{1.1}$$

$$\overline{SD} = \frac{OH}{a_\nu - a_1} \tag{1.2}$$

Reducing operating hours directly translates into energy and cost savings. [26] showed that the energy consumption of an idling server is sometimes equal to one running on a 50 % utilization level. CPU utilization serves as a robust indicator for a server's energy consumption that can be predicted by Equation 1.3 [26], $u \in [0, 1]$ being the CPU utilization, and P the energy demand in watts.

$$P_{idle} + (P_{busy} - P_{idle}) \cdot u \tag{1.3}$$

Tests on one of our own servers (2 Intel Xeon CPUs, 64 GB RAM, 6 disks,

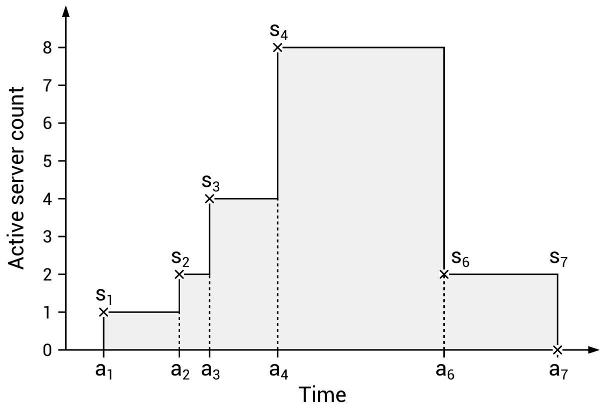


Figure 1.1: A data point (a_n, s_n) is recorded each time a_n the server demand s_n is changed

2 power supply units (PSUs)) confirmed Equation 1.3. Through the server's management console we obtained power consumption values for $P_{idle} = 160\,\mathrm{W}$ when idling and $P_{busy} = 270\,\mathrm{W}$ when solving mixed integer programs at 100 % CPU utilization.

Based on Equation 1.3, energy consumption at $u = 0.30$ is 193 W. Consolidating two such servers to a single one with an aggregated load of $u = 0.60$ would yield energy savings of forty percent (41 %) as total energy consumption drops from 386 W to 226 W.

[26] found that for CPU intensive workloads, DVS can decrease a server's energy demand by more than 20 % depending on the application and the aggressiveness of the algorithm. For our scenario, we assume a very aggressive DVS configuration with a 50 % reduction on CPU's energy demand (see Equation 1.4 [26]). Both servers with $u = 0.3$ will now consume 176 W each, a saving of 8 % due to DVS. DVS cannot be applied to the consolidated server as its utilization is above 50 %. Still, consolidating those two servers with active DVS to a single one without DVS reduces energy consumption by 36 %.

$$P_{idle} + \frac{1}{2} \cdot (P_{busy} - P_{idle}) \cdot u \tag{1.4}$$

We conclude that energy savings achieved by reducing server operating hours are considerable, even if techniques like DVS are taken into account. P_{idle} is especially important in this calculation: with lower values for P_{idle} consolidation becomes less effective as additional energy savings through consolidation decrease due to a decreasing P_{idle}. If all devices of a server had a low energy consumption when idling (i.e. $P_{idle} = 0 + \varepsilon$) and their energy demand scales linearly with their utilization, server consolidation would be without effect. However, this is not the case for server architectures found today. Consolidation is a promising option for data center operators to reduce energy demand and operating costs.

In all considered scenarios, our primary objective is to allocate VMs efficiently to servers. We will measure or simulate the average server demand $\overline{SD}$. To compare outcomes of different scenarios, like ones with unequal number of VMs, we will use the term **allocation density**. It is calculated by $d = {}^{n}\!/\!_{m}$ with n denoting the number of VMs and m the number of required servers.

Allocation density often cannot be increased without affecting service level objectives (SLOs) at some point. Our goal is to increase allocation density while maintaining a high service quality above 99 %. In experiments, service quality is measured by tracking application request response times. Requests with a response time between $[3\,\mathrm{s}, \infty[$ are considered as failed. Service quality is calculated as the percentage of successful application requests.

Allocation quality describes the combination of **allocation density** and **service quality**. Therefore, our primary objective is a high allocation quality by increasing allocation density and service quality.

Part I

Allocation of persistent services

Chapter 2

Static vs. dynamic controllers

In this work we focus on enterprise clouds hosting long-running transactional applications in VMs. A central managerial goal in IT service operations is to minimize server operating hours while maintaining a certain service quality, in particular response times. In the literature, this problem is referred to as server consolidation [86, 77] or workload concentration [51].

This is a new type of capacity planning problems, which is different from the queuing theory models that have been used earlier for computers with a dedicated assignment of applications [83]. Server consolidation is also different from workload scheduling where short-term batch jobs of a particular length are assigned to servers [14]. Workload scheduling is related to classical scheduling problems, and there is a variety of established software tools such as the IBM Tivoli Workload Scheduler, LoadLeveler or the TORQUE Resource Manager. In contrast, workload consolidation through VM allocation deals with the assignment of long-running VMs with a seasonal resource demand to servers. Consequently, the optimization models and VM allocation mechanisms are quite different. Overall, workload consolidation can be seen as a new branch in the literature on capacity planning and resource allocation in operations research.

VM consolidation can be implemented by static controllers that calculate a static allocation of VMs to servers over time [61, 71, 79, 86]. Based on the workload profiles of VMs an allocation to servers is computed such that the total number of servers is minimized. A profile describes the seasonal resource utilization of a VM over an average business day. Such an environment is different from public clouds, where VMs are sometimes reserved for short amounts of time only, or where some applications exhibit an unpredictable resource demand. Among the vast majority of applications that run in enterprises, such applications are the exception rather than the rule.

At the core of static controllers are high-dimensional NP-complete bin packing problems. Computational complexity is a considerable practical problem. Recent algorithmic advances allow to solve very large problem sizes with several hundred VMs using a combination of singular value decomposition (SVD) and integer programming techniques [77].

VM migrations can be leveraged to optimize the VM allocation continuously to decrease server operating hours. Some platforms such as VMware vSphere, OpenNebula[1], or Ovirt[2] provide virtual infrastructure management that allocates VMs dynamically to servers. They closely monitor the server infrastructure to detect resource bottlenecks by thresholds. If such a bottleneck is detected or expected to occur in the future, they take actions to dissolve it by migrating VMs to different servers. We will refer to such techniques as dynamic controllers as opposed to static controllers.

Many managers of corporate clouds consider moving to dynamic controllers [62] and there are various products available by commercial or open-source software providers. Also, several academic papers on virtual infrastructure management using dynamic controllers illustrate high server demand savings [51, 8, 78, 3, 33, 22]. When hosting long-running business applications in corporate clouds, dynamic controllers promise autonomic VM allocation with no manual intervention and high allocation density due to the possibility to immediately respond to changes in server utilization.

For IT service managers it is important to understand if, and how much, dynamic controllers can save in terms of average server demand compared to

[1]opennebula.org
[2]ovirt.org

static controllers. In this work, we want to address the question – *Should managers rely on dynamic allocations or rather use static allocations with optimization-based algorithms for the VM allocation in private clouds with long-running transactional business applications?* Surprisingly, there is little research guiding managers on this question (see Section 2.1).

Much of the academic literature is based on simulations, where the latencies, migration overheads, and the many dependencies of VMs, hypervisors, server hardware, and network are difficult to model. The external validity of such simulations can be low. Therefore, experiments are important for the external validity of results. The set-up of a testbed infrastructure including the hardware, workload generators, management, and monitoring software is time consuming and financially expensive, which might explain the lack of experimental research results to some degree.

The main contribution of this work is an extensive experimental evaluation of static and dynamic VM allocation controllers. We implemented a testbed infrastructure with physical servers and a comprehensive management and monitoring framework. We used widely accepted applications such as SPECjEnterprise2010 to emulate real-world business applications. Workload demand was based on a large set of utilization traces from two European IT service providers.

Our goal is to achieve external validity of the results, but at the same time maintain the advantages of a lab environment, where the different allocation controllers can be evaluated and experiments can be analyzed and repeated with different workloads. Our experiments analyze different types of static and dynamic allocation controllers including pure threshold-based ones, which are typically found in software solutions, but also ones that employ forecasting.

Our main result is that with typical workloads of business applications, static controllers lead to higher allocation density compared to dynamic ones. This is partly due to resource utilization overheads and response time peaks caused by VM migrations. The result is robust with respect to different thresholds, even in cases where the workloads are significantly different form the expected ones.

Even though the overhead caused by VM migrations has been discussed [28], the impact on different VM allocation controllers has not been shown so far. Still, it is of high importance to IT service operations. VM migration algorithms have become very efficient and the main result of our research carries over to other hypervisors. Mainly, because memory always needs to be transferred from one server to another.

Our simulations use the same allocation controller implementations as used in the testbed. We took great care to reflect system-level particularities found in the experiments. Interestingly, the efficiency of static controllers increases for larger environments with several hundred VMs. They possess more complementarities in workloads that can be used to allocate VMs more densely. The result is a clear recommendation to use optimization for VM allocation. VM migrations should be used in exceptional cases, e.g. for server maintenance tasks.

Texts in this chapter are based on a previous publication [98].

2.1 Related work

In what follows, we will revisit the literature on static and dynamic allocation controllers.

2.1.1 Static allocation controllers

Research on static allocation controllers assumes that the number and workload profiles of servers are known, which turns out to be a reasonable assumption for the majority of applications in enterprise data centers [77, 61, 71, 79, 86]. For example, email servers typically face high loads in the morning and after the lunch break when most employees download their emails; payroll accounting is often performed at the end of the week; data warehouse servers experience a daily peak load early in the morning when managers access their reports.

The fundamental problem is to assign VMs with seasonal workloads to servers such that the total number of servers gets minimized without overloading them. For example, [86] showed that a static allocation considering daily workload seasonality can lead to 30 % - 35 % savings in server demand compared to simple heuristics based on peak workload demands. The fundamental problem described in the above papers can be reduced to the multidimensional bin packing problem, a known NP-complete optimization problem. This particular problem can be solved by mathematical optimization. However, such optimization approaches often do not scale to real-world problem sizes with hundreds of VMs and servers.

A recent algorithmic approach combining SVD and integer programming allows to solve large instances of the problem [77]. In this work, we will use optimization models from [86] and [77]. In contrast to earlier work, we actually deploy the resulting VM allocations on a testbed infrastructure such that the approach faces all the challenges of a real-world implementation. This is a considerable extra effort beyond simulations only, but it provides evidence of the practical applicability.

2.1.2 Dynamic allocation controllers

The VM migration technology is supported by all major hypervisors. It allows migrating a VM during runtime from one server to another. The algorithm first copies the working memory of the VM to the target server. In parallel it tracks memory write operations and marks all memory pages that got altered after copying them to the target server. These pages are re-transferred subsequently. VM migrations require additional CPU and network capacity [1].

Migration technology enables dynamic controllers as an alternative to static ones. All commercial and open-source approaches that we are aware of rely on some sort of threshold-based approach. It monitors the server infrastructure and is activated if certain resource utilization thresholds are exceeded. VMs are migrated between servers in order to mitigate these violations. VMware's Distributed Resource Management [37] and Sandpiper [101] are examples for such systems. Both, [37] and [3] motivate the need for workload prediction in order to avoid unnecessary migrations. In our experiments, we use both, simple threshold-based or reactive implementations and such that employ forecasting.

2.2 Experimental infrastructure

We will now briefly discuss the allocation controllers we implemented and studied by experiments and simulations.

2.2.1 VM allocation controllers

We will distinguish between several static and dynamic controllers. Round robin and optimization are two static controllers. They are executed once at the beginning of an experiment where they calculate a static allocation of VMs to servers. Both, reactive and proactive are dynamic controllers. The reactive one acts on resource utilization thresholds solely while the proactive one leverages forecasting techniques to detect server overloads and underloads.

2.2.1.1 Round robin controller

The round robin controller is a heuristic to create a static allocation. It should serve as an example for a heuristic as typically used in practice. First, a number of required servers is determined by adding the maximum resource demands of all VMs and then dividing by the server capacities. This is done for each resource individually. The number of required servers is rounded to the next integer. Then the VMs are distributed in a round robin manner to the appropriate number of servers so that the available memory is not oversubscribed.

2.2.1.2 Optimization and overbooking controllers

We used the static server allocation program with variable utilization (SSAPv)[86] to compute an optimal static server allocation. We will briefly introduce the corresponding mixed integer program formulation, which is also the foundation for the algorithms used in [77].

$$
\begin{aligned}
&\min \sum_{i=1}^{I} y_i \\
&\text{s.t.} \\
&\sum_{i=1}^{I} x_{ij} && = 1, && \forall j \in J \\
&\sum_{j=1}^{J} u_{jkt} x_{ij} && \leq s_{ik} y_i, && \forall i \in I, \forall k \in K, \forall t \in \tau \\
&y_i, x_{ij} \in \{0,1\}, && && \forall i \in I, \forall j \in J
\end{aligned}
\tag{2.1}
$$

The program in Equation 2.1 assigns a set of VMs $j \in J$ to a set of servers $i \in I$, while considering multiple resources $k \in K$ like CPU or memory over a discrete time horizon of $t \in \tau$ periods. A server's size is denoted by s_{ik} describing its capacity for each resource k, e.g. 200 for the CPU of a quad-core server. $y_i \in \{0,1\}$ tells whether a server i is active. A server is active if it is hosting at least one VM. The allocation matrix x_{ij} indicates whether VM j is assigned to server i. u_{jkt} describes the utilization of VM j for resource k at time t such as for CPU with values between 0 and 100 for a dual-core VM. The objective function minimizes the overall server demand.

The first set of constraints ensures that each VM is allocated to exactly one server. The second set of constraints ensures that the aggregated resource demand of multiple VMs does not exceed a server's capacity per server, period, and resource.

The optimization model was implemented using the Gurobi branch and cut solver. It requires the resource capacity of all servers and the workload profiles u_{jkt} of all VMs as an input. For the experiments, the parameters were set in accordance with the hardware specification of the testbed infrastructure. Workload profiles from two European IT service providers were used to calculate the allocations (see Section 2.2.2 for more details).

In scenarios with overbooking, server resource capacities s_{ik} were increased beyond the actual server capacity. For our experiments server CPU capacity was overestimated by 15 % (230), a value that was determined by experimentation. This accounts for the reduction of variance by adding multiple VM resource demands (random variables) and leads to higher utilization with little impact on the service level, if overbooking is at the right level.

For larger problem instances, the optimization model could not be solved any more as the large number of capacity constraints and dimensions to be considered renders this task intractable. Here, we refer to a dimension as the utilization of a resource by a VM in a period, i.e., an unique tuple (k, t) corresponding to a particular row in the constraint matrix.

[77] describes an algorithm based on truncated SVD which allows solving larger problems as well with near-optimal solution quality. Details of the algorithm are explained in [77]. An evaluation of the SVD-based approach using workload data from a large data center has shown that this leads to a high solution quality. At the same time it allows for solving considerably larger problem instances of hundreds of VMs than what would be possible without SVD. In our simulations, we apply this approach to derive static allocations for lager scenarios with up to 360 VMs.

2.2.1.3 Reactive controller

The reactive controller is of dynamic nature and migrates VMs so that the number of servers gets minimized and server overload situations are avoided. A migration is triggered if the utilization of a server exceeds or falls below a certain threshold. It balances the load across all servers similar to the mechanisms described by [101]. Algorithm 1 illustrates the actions taken in each control loop.

```
Data: Servers $S$ and VMs $V$
function CONTROL(S, V):
    $\hat{S}$ ←FIND-VIOLATED-SERVERS($S$)
    UPDATE-VOLUME-VSR($S$, $V$)
    forall the $s \in \hat{S}$ do
        $\hat{V}$ ←$\{v \in V | \mathsf{v.s} = s\}$
        $\hat{V}$ ←sort($\hat{V}$, order=DEC, by=vsr)
        forall the $v \in \hat{V}$ do
            t ←FIND-TARGET($v$, $\{t \in S | \mathsf{t.active}\}$)
            if t = $NULL$ then
                t ←FIND-TARGET($v$, $S$)
            end
            MIGRATE($s$, $v$, $t$, 30)
        end
    end
end
```

Algorithm 1: Working of the reactive controller

The controller uses the Sonar monitoring system as described in Chapter B to receive the CPU and memory utilization of all servers and VMs in 3 s intervals. Utilization measurements are stored for 10 min. Overload and underload situations are detected by a control loop that runs every 5 min.

The function FIND-VIOLATED-SERVERS marks a server as overloaded or underloaded if $M = 17$ out of the $K = 20$ recent CPU utilization measurements are above or below a given threshold $T_{overload}$ or $T_{underload}$. An underload threshold of 40 % and an overload threshold of 90 % were chosen based on preliminary tests as described in Section 2.4.4.

$$\begin{aligned} VOL &= \frac{1}{1-cpu} \cdot \frac{1}{1-mem} \\ VSR &= \frac{VOL}{mem} \end{aligned} \tag{2.2}$$

Overloaded and underloaded servers are marked and handled by offloading a VM to another server. A VM on a marked server has to be chosen in conjunction with a migration target server. Target servers are chosen based on their VOL. VMs are chosen based on their VSR ranking which prioritizes VMs with a low memory demand but high VOL. Both, the server VOL and VM VSR values are calculated by the function UPDATE-VOLUME-VSR following Equations 2.2 as proposed by [101].

```
function FIND-TARGET(v, S):
    if v.s.mark = OVRL then
        S ←sort(S, order=ASC, by=vol)
    else
        S ←sort(S, order=DEC, by=vol)
    end
    forall the {$s \in S|\neg$s.blocked} do
        lsrv ← ptile(s.load[$-K : 0$], 80)
        lvm ← ptile(v.load[$-K : 0$], 80)
        if (lsrv + lvm) < $T_{overload}$ then
            return $s$
        end
    end
end
```
Algorithm 2: Find target server procedure in the reactive controller

The algorithm tries to migrate VMs away from the marked server in descending order of their VSR. For each VM in this list, the algorithm described by function FIND-TARGET in Algorithm 2 searches through the server list to find a migration target server. For overloaded servers, migration target servers with low VOL are considered first. Target servers with a high VOL are considered first for underloaded source servers. A server is a viable migration target if the 80th percentile of its K most recent utilization measurements plus the ones of the VM are below the overload threshold and if the target server is not blocked from previous migrations.

Only one migration is allowed at a time for each server, either an incoming or outgoing one. The migration process itself consumes CPU and memory resources. Resource utilization measurements used to decide about triggering migrations must not be influenced by this overhead. Therefore, servers involved in a migration are blocked for 30 s after the end of a migration. This block time is passed as a parameter to the MIGRATE function. Subsequently they are re-evaluated for overload and underload situations. For a similar reasons, the controller halts its execution during the first 2 min of its execution to fill its resource utilization measurement buffers.

For simulations and experiments, an initial static allocation was calculated by the optimization controller as described in Section 2.2.1.2.

2.2.1.4 Proactive controller

The proactive controller extends the reactive one by a time series forecast to avoid unnecessary migrations. A migration will only be triggered if the forecast suggests that the overload or underload is persistent and not only driven by an unforeseen demand spike. A forecast is computed if a threshold violation is detected using double exponential smoothing (DES) with values $\alpha = 0.2$ and $\gamma = 0.1$.

We evaluated different forecasting methods such as autoregressive (AR) models, using mean as forecast, or simple exponential smoothing [39], but DES came out best (see Section 2.4.4). As the differences among several forecasting techniques on the allocation density were small, we will only report on those experiments with DES.

The proactive controller extends the reactive one slightly by modifying the function FIND-VIOLATED-SERVERS as shown in Algorithm 3. For each server a load forecast is computed using 1 min of utilization measurements. If the forecast and M out of K measurements pass a threshold, an overload or underload situation is detected and the server gets marked.

2.2.2 Workload

We leveraged a set of 451 server utilization time series from two large European IT service providers. The time series contain CPU and memory usage

```
function FIND-VIOLATED-SERVERS(S):
    s.mark = 0, $\forall s \in S$
    forall the $s \in S$ do
        // For proactive controller only
        lfcst ← forecast(s.load)

        // # of measurements above or below a threshold
        ucrt ← $|$s.load$[-K : 0] < T_{underload}|$
        ocrt ← $|$s.load$[-K : 0] > T_{overload}|$

        if ocrt $> M$ and lfcst $> T_{overload}$ then
            s.mark = OVRL
        else if ucrt $> M$ and lfcst $< T_{underload}$ then
            s.mark = UNDL
        end
    end
    return $\{s \in S|$s.mark $\neq 0\}$
end
```

Algorithm 3: Forecast in the proactive controller to detect overloaded and underloaded servers

in a sampling rate of 5 min over a duration of 10 weeks for SET1 and web service request throughput at a sampling rate of 60 min over a duration of 77 days for SET2. All servers were running enterprise applications like enterprise resource planning (ERP) and web, application, and database (WAD) services. Autocorrelation functions showed that seasonality on a daily and weekly basis is present in almost all time series as it has also been found in related works [34, 86]. Appendix E provides an extensive statistical analysis over all time series.

We sampled three distinct sets (MIX1, MIX2, MIX3) with 18 time series each to model different but realistic VM workload profiles. The first set contains time series with low variance, while the second one has time series with high variance and many bursts. The third set was generated by randomly sampling nine time series without replacement from the first two sets. It meant to cover scenarios with both, predictable and unpredictable workloads.

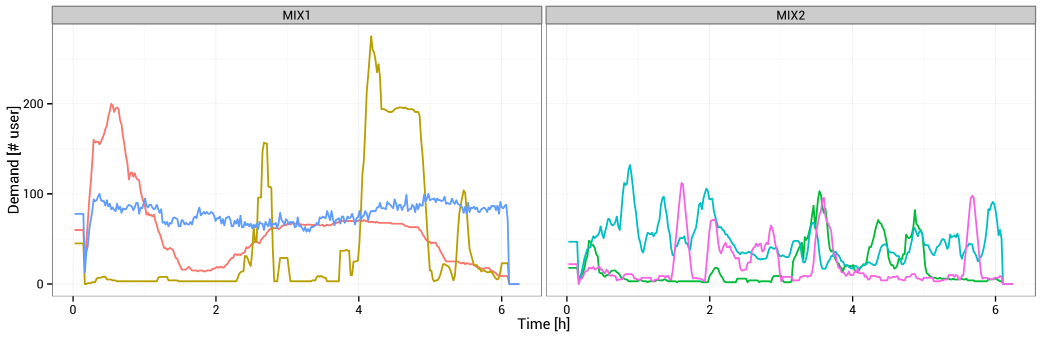


Figure 2.1: Workload profile samples for MIX1 and MIX2

Based on the selected time series we generated workload profiles by extracting the average resource utilization for one day. Here, we followed the approach described in Appendix E which is based on [86] and [32]. Samples of MIX1 and MIX2 are shown in Figure 2.1. The shape of the MIX1 workload profiles does not show short-term bursts or random jumps. However, there can be an increased demand in the morning, evening, or during the operational business hours of a day compared to historical workloads. MIX2 in contrast exhibits peaks and is not as smooth as MIX1.

According to the extracted workload profiles we generated workload on a target VM that was running on a dedicated server with the SPECjEnterprise2010 application using the Rain workload generator. During the benchmark we monitored the VM's resource utilization which was then used to parametrize the optimization models of static controllers.

The measured utilization traces describe the VM utilization in a range of $[0, 100]$ on each logical CPU. Each VM uses 2 virtual and physical CPU cores while a server has 4 physical CPU cores. Therefore, we parametrized the CPU capacity of one server as $s_{i\ \mathrm{CPU}} = 200$ capacity units in the default configuration and $s_{i\ \mathrm{CPU}} = 230$ units for the overbooking configuration.

In addition, we conducted a second set of experiments where we took the workload mixes MIX1-3 to determine an allocation, but added noise to the workload profiles which were then used by the Rain workload generators during experiments. This generates a scenario, where the demand and consequently

Metric	MIX1	MIX2	MIX3
$mean(\bar{x_0}, .., \bar{x_n}) - mean(\bar{y_0}, .., \bar{y_n})$	4.29	5.57	5.58
$mean(p^{50}_{x_0}, .., p^{50}_{x_n}) - mean(y^{p50}_0, .., y^{p50}_n)$	2.60	5.40	5.05
$mean(p^{90}_{x_0}, .., p^{90}_{x_n}) - mean(y^{p90}_0, .., y^{p90}_n)$	15.61	13.71	11.68
$mean(\sigma_{x_0}, .., \sigma_{x_n}) - mean(\sigma_{y_0}, .., \sigma_{y_n})$	6.46	6.01	4.94
$mean(corr(x_0, y_0), .., corr(x_n, y_n))$	0.29	0.46	0.33

Table 2.1: Pairwise comparison of the workload profiles for MIX1-3 with the corresponding ones of MIX1-3m. All workload profiles in the default mix are depicted by x_i and y_i is used for the modified workload profiles. The 50th percentile of a workload profile is indicated by $p^{50}_{x_i}$.

the workload traces differ considerably from those used to compute the static allocation, and describe a challenging scenario for static controllers.

Each workload profile was changed by scaling the values linearly using factors $[0.8, 1.3]$ and shifting it by $[-30, +30]$ minutes. Shifting did not alter its length. Elements moved beyond the workload profiles end were re-inserted at its beginning.

Table 2.1 describes difference between the default and modified workload profiles for MIX1-3 versus MIX1m-3m. Different statistics like mean or median were calculated over all traces and the average is taken. Modified workloads contain more peak demands as shown by the 90th percentile. Spearman correlation coefficient shows slight similarities for MIX1 and MIX3 with their modified counterparts. There is a higher correlation for MIX2 which is mostly due to the volatile nature of the workload profiles.

2.3 Experimental design and procedures

We analyzed five different controllers with the six workload mixes (MIX1-3 and MIX1m-3m) described in the previous section. During an experiment a number of core-metrics was recorded. The number of VMs was not varied between the experiments, nor were the threshold levels of the dynamic controllers varied. Similar to real-world environments, the settings for reactive and proactive controllers were chosen based on preliminary tests with the expected workload which are described in Section 2.4.4.

Apart from the experiments, we also run simulations with a larger number of servers and VMs to see, if the results carry over to larger scenarios. We took great care to run the simulations such that the same migration overheads observed in the lab were taken into account.

The detailed interactions of the SPECjEnterprise2010 application and Java container or database were not covered by the simulation framework. Instead the resource demands were added at a particular point in time. For this reason, we will only report the number of servers used, the number of CPU overloads, and the number of migrations. CPU overloads are calculated by counting the number of simulation intervals where the resource demand of VMs is beyond the capacity of a server. Obviously, simulations do not have the same external validity than experiments, but they can give an indication of the savings to be expected in larger environments like data centers.

In an initialization phase 18 VMs were cloned from a template with Glassfish and MySQL services installed. The initial allocation was computed and the VMs were deployed on the servers accordingly. All VMs were rebooted to reset the operating system and to clear application caches. A setup process as described in Chapter C started the Glassfish and MySQL services, loaded a database dump, configured the Rain load generator with the selected workload profiles and finally triggered Rain to generate load against the VMs. This setup phase was followed by a 10 min ramp-up phase during which the Rain generators establish their initial connections to Glassfish and generate a moderate workload that is equal to the first minute of the workload profile. Then the reactive or proactive controllers were started and the workload profile was replayed by Rain.

Sonar was used to monitor the relevant utilization level on all VMs and servers in 3 s intervals. Each experiment took 6 h, where all relevant resources such as CPU, memory, disk, and network were monitored. Additionally all Rain generators reported 3 s averages of the response time for each service individually. This allows a complete replication of a benchmark for analytical purposes. The accumulated net time of all experiments reported exceeds 50 days.

2.4 Results

In the following we will describe the results of our experiments on the testbed infrastructure as well as the results of simulations to study our controllers in larger scenarios.

2.4.1 Experiments with original workload mix

First, we will describe the experimental results with the workloads of MIX1-3. We will mainly report aggregated metrics such as the average and maximum response time of the services, operations per second, the number of response time violations, and the number of migrations of each 6 h experiment for all VMs and applications. The values in Table 2.2 are averages of three runs of a 6 h experiment with identical treatments. Due to system latencies there can be differences between these runs. Values in parentheses describe the variance and values in squared brackets describe the min and max values over 3 replications. Violations state the absolute number of 3 s intervals, where the average response time over all request was beyond the threshold of three seconds. The service quality indicates the percentage of these intervals without violations.

Across all three workload sets the static controller with overbooking had the lowest number of servers on average. This comes at the expense of increased average response times compared to other static controllers. The maximum response time was worse for reactive systems throughout. Almost all controllers achieved a service quality of 99 % except for proactive (MIX1) with 98.46 % and overbooking (MIX2) with 97.85 %.

The results of the optimization controller were comparable to dynamic controllers in terms of server hours, sometimes better (MIX2), sometimes worse (MIX0 and MIX1). The average response times of the optimization controllers were always lower than those of the dynamic ones.

Dynamic controllers come at the expense of migrations, which static controllers only have in exceptional cases such as manually triggered emergency migra-

Controller	$\overline{\text{Srv}}$	$\overline{\text{CPU}}$	$\overline{\text{RT}}$	$\lceil\text{RT}\rceil$	$\frac{\overline{\text{O}}}{[\text{sec}]}$	I_{late}	O_{fail}	Mig	SQ
				MIX 1					
Round Robin	6 (0)	69.29	352	20186 (972)	151 (0)	138 (6)	12 (2)	0 [0/0]	99.89%
Optimization	6 (0)	68.64	330	17621 (3777)	151 (0)	137 (26)	8 (2)	0 [0/0]	99.89%
Underbooking	6 (0)	68.64	330	17621 (3777)	151 (0)	137 (26)	8 (2)	0 [0/0]	99.89%
Overbooking	5 (0)	78.96	466	19103 (2811)	149 (0)	647 (181)	10 (4)	0 [0/0]	99.5%
Proactive	5.95 (0.07)	73.94	566	42012 (5958)	147 (2)	1990 (1609)	15 (5)	10.33 [9/12]	98.46%
Reactive	6 (0)	69.62	392	21501 (9386)	150 (1)	279 (25)	14 (1)	0.33 [0/1]	99.78%
				MIX 2					
Round Robin	6 (0)	38.09	388	17016 (15823)	81 (0)	289 (11)	5 (1)	0 [0/0]	99.78%
Optimization	4 (0)	55.03	467	16875 (7071)	80 (1)	637 (156)	6 (4)	0 [0/0]	99.51%
Underbooking	4 (0)	56.50	459	18464 (6179)	80 (0)	552 (240)	5 (4)	0 [0/0]	99.57%
Overbooking	3 (0)	70.88	744	34498 (7538)	77 (0)	2783 (106)	5 (3)	0 [0/0]	97.85%
Proactive	3.93 (0.2)	58.43	535	65337 (22243)	79 (0)	777 (184)	22 (17)	23 [16/34]	99.4%
Reactive	4.34 (0.18)	54.25	547	71153 (23498)	79 (1)	842 (359)	28 (23)	26.4 [18/36]	99.35%
				MIX 3					
Round Robin	6 (0)	49.27	377	12590 (2698)	107 (0)	111 (13)	8 (2)	0 [0/0]	99.91%
Optimization	5 (0)	56.42	347	11222 (1171)	107 (0)	73 (6)	8 (2)	0 [0/0]	99.94%
Underbooking	5 (0)	56.42	347	11222 (1171)	107 (0)	73 (6)	8 (2)	0 [0/0]	99.94%
Overbooking	4 (0)	68.04	483	21387 (1515)	106 (0)	673 (143)	8 (2)	0 [0/0]	99.48%
Overbooking^2	3 (0)	86.91	990	29887 (0)	98 (0)	3916 (0)	11 (0)	0 [0/0]	96.98%
Proactive	4.76 (0.16)	62.98	475	54636 (215)	106 (0)	545 (93)	12 (4)	14.33 [10/17]	99.58%
Reactive	4.85 (0.12)	62.74	505	59651 (9129)	105 (1)	635 (158)	19 (11)	17 [17/17]	99.51%

Table 2.2: Experimental results on static vs. dynamic VM allocation controllers. $\overline{\text{Srv}}$ – average server demand, $\overline{\text{CPU}}$ [%] – average CPU utilization, $\overline{\text{RT}}$ [ms] – average response time, $\lceil\text{RT}\rceil$ [ms] – maximum response time, $\frac{\overline{\text{O}}}{[\text{sec}]}$ – average operations per second, I_{late} – 3 s intervals with $\overline{\text{RT}} > 3$ s, O_{fail} – failed operations count, Mig – VM migration count, SQ [%] – service quality based on 3 s intervals

tions. For all experiments the total number of migrations was below 37 per experiment. On average a migration was triggered every 3 hours per VM.

The proactive controller with time series forecasting led to a slightly lower number of servers and migrations compared to reactive one in case of MIX2 and MIX3 but triggered much more migrations for MIX1.

The variance of the average response time among the three identical experimental runs increased for the dynamic controller compared to the static ones. Even minor differences in the utilization can lead to different migration decisions and influence the results. This seems to be counteracted by the proactive controller which was more robust against random load spikes due to its time series forecasting mechanism.

We used a Welch test to compare the differences in the response times of the different controllers at a significance level of $\alpha = 0.05$. All pairwise comparisons for the different controllers and mixes were significant, except for the difference of proactive and overbooking (MIX3).

Overall, dynamic controllers did not lead to a significantly higher allocation quality compared to optimization-based static controllers. A number of factors exist which can explain this result. One is the additional overhead of migrations which can also lead to additional response time violations. This overhead might compensate advantages one would expect from dynamic controllers.

Some of the migrations of the reactive controller were triggered by short demand peaks and proof unnecessary afterward. One could even imagine situations, where a controller migrates VMs back and forth between two servers as their workload bounces around the threshold levels. A proactive controller with some forecasting capabilities can filter out such demand spikes in order to avoid unnecessary migrations.

2.4.2 Experiments with modified workload mix

We wanted to see, if the results for the workload sets MIX1-3 carry over to a more challenging environment, where the actual workload during an experiment differs considerably from those used to compute a static allocation. The modified demand traces of the sets MIX1m-3m were used by the Rain workload generators while the static allocation was still computed with the original profiles MIX1-3. For this reason, the average number of servers remained the same as for the first experiments for all static controllers. The results of these second experiments are described in Table 2.3.

One would expect that static controllers are much worse in such an environment compared to their dynamic counterparts. Surprisingly, the main result carries over. The service qualities were high and average response times were low in all treatments.

Again, we used a Welch test to compare the differences in the response times of the different controllers at a significance level of $\alpha = 0.05$. All pairwise comparisons for the different controllers and mixes were significant, except for overbooking to proactive in MIX1m ($p = 0.01$), reactive to proactive in MIX2m ($p = 0.87$), and optimization to reactive in MIX3M ($p = 0.14$). For overbooking in MIX2m and MIX3m an increased average and maximum response time with a service quality degradation to 91.74 % and 96.37 % was observed.

Dynamic controllers showed a service quality degradation for MIX1m with 96.81 % for the reactive and 97.74 % for the proactive controller. This can be explained by the overall workload demand, which is close to what the six servers were able to handle. The average server utilization was near 80 % over the complete six hours and all servers. As a result average response times increased for all controllers compared to the first experiments. In this case even slightly suboptimal allocations result in a degradation of service quality during periods of high utilization which especially affects the overbooking and dynamic controllers. The optimization controller in contrast still achieved a good service quality above 99 % and comparably low average response times.

Comparing the throughput in operations per second with the first experiments shows an increase for MIX1m-3m. This was an expected result as all modified workload profiles entail an increased demand as shown in Table 2.1.

For MIX2m-3m the dynamic controllers showed a similar behavior as for MIX2-3. The average response time remained constant while the maximum response times were increased compared to static controllers. Both controllers were able to maintain a service quality above 99 % by an increased average server demand.

Dynamic controllers triggered the same number or more migrations compared to the first experiments. However, for MIX2m, the volatile workload scenario, the migration counter of the reactive controller was substantially increased with 45 migrations on average while the proactive controller required only 22 migrations. The variance in the average response time tends to be higher for dynamic controllers. Overall, even in scenarios where workload volatility increases for all VMs, the static optimization-based allocations still perform well.

Another working paper of our group describes a set of experiments on the same hardware, but with an entirely different software infrastructure, different hypervisor (Citrix XenServer), different reactive controller, different operating systems, and a different workload generator [88]. While the infrastructure was less stable and focused on the evaluation of reactive control parameters, also these initial experiments found that the static allocation and a modest level of overbooking yielded a high allocation density and response times compared to reactive ones. These initial experiments used $T_{Underload}$ thresholds of 20 % and 30 % and $T_{Overload}$ thresholds of 75 % and 85 % for the reactive controller. However, efficient thresholds depend on the workload and determining good values is certainly not an easy task for IT service managers. Overall, this provides some evidence that our main result carries over to different implementations of the reactive controller, the thresholds used, or different samples of the workload.

Controller	$\overline{\text{Srv}}$	$\overline{\text{CPU}}$	$\overline{\text{RT}}$	$\lceil\text{RT}\rceil$	$\frac{\overline{\text{O}}}{[\text{sec}]}$	I_{late}	O_{fail}	Mig	SQ
MIX 1 MOD									
Round Robin	6 (0)	73.96	449	25087 (2911)	165 (0)	829 (105)	10 (3)	0 [0/0]	99.36%
Optimization	6 (0)	73.75	440	27080 (6945)	165 (0)	983 (81)	11 (5)	0 [0/0]	99.24%
Overbooking	5 (0)	82.92	618	27247 (4516)	160 (3)	2012 (369)	49708 (86071)	0 [0/0]	98.45%
Proactive	5.96 (0.07)	76.44	600	55370 (28537)	162 (2)	2928 (828)	858 (1692)	7.75 [4/13]	97.74%
Reactive	5.99 (0)	79.33	710	47025 (17355)	160 (0)	4133 (787)	20 (3)	14.33 [12/18]	96.81%
MIX 2 MOD									
Round Robin	6 (0)	46.82	375	13717 (5646)	101 (0)	368 (38)	5 (1)	0 [0/0]	99.72%
Optimization	4 (0)	65.04	441	20766 (8736)	101 (0)	366 (32)	6 (1)	0 [0/0]	99.72%
Overbooking	3 (0)	81.04	1401	60584 (11310)	90 (0)	10699 (128)	64 (95)	0 [0/0]	91.74%
Proactive	4.79 (0.03)	61.02	511	82047 (26877)	99 (0)	807 (127)	31 (21)	22 [20/25]	99.38%
Reactive	4.94 (0.05)	61.98	545	78349 (15210)	98 (1)	1095 (264)	338 (586)	45 [40/50]	99.16%
MIX 3 MOD									
Round Robin	6 (0)	58.77	382	18623 (4912)	128 (0)	179 (34)	10 (2)	0 [0/0]	99.86%
Optimization	5 (0)	67.80	486	25757 (3737)	127 (0)	1290 (83)	11 (2)	0 [0/0]	99%
Overbooking	4 (0)	77.63	823	31582 (1571)	123 (0)	4706 (200)	11 (2)	0 [0/0]	96.37%
Proactive	5.5 (0.16)	65.83	465	49300 (13668)	127 (1)	802 (415)	19 (5)	18 [12/24]	99.38%
Reactive	5.6 (0.03)	67.51	485	74017 (16339)	126 (1)	774 (317)	27 (18)	23.67 [18/33]	99.4%

Table 2.3: Experimental results for mixes MIX1m-3m. $\overline{\text{Srv}}$ – average server demand, $\overline{\text{CPU}}$ [%] – average CPU utilization, $\overline{\text{RT}}$ [ms] – average response time, $\lceil\text{RT}\rceil$ [ms] – maximum response time, $\frac{\overline{\text{O}}}{[\text{sec}]}$ – average operations per second, I_{late} – 3 s intervals with $\overline{\text{RT}} > 3$ s, O_{fail} – failed operations count, Mig – VM migration count, SQ [%] – service quality based on 3 s intervals

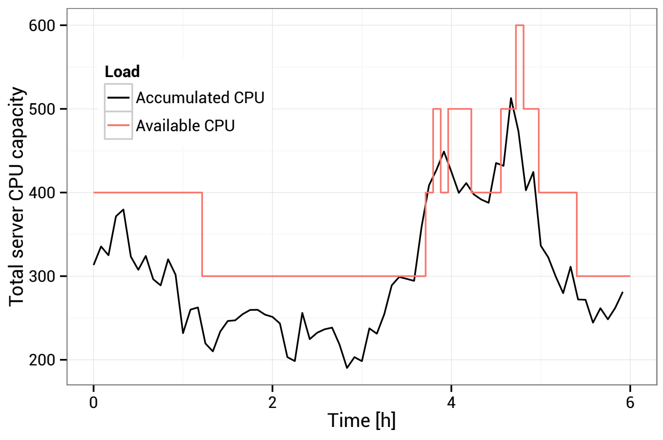


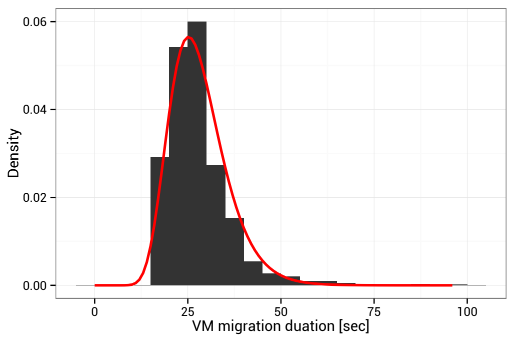
Figure 2.2: Accumulated load of six servers with the number of active servers as allocated by the proactive controller

Note that the results do not hold for all data centers and workload types. For example, in regional data centers, where all business applications exhibit a very low utilization at night time, it can obviously save additional servers to consolidate the machines after working hours. As shown in Figure 2.2 servers can efficiently be evacuated during night times (between 1 h to 3 h). Such nightly workload consolidation can be triggered automatically and in addition to static controllers.

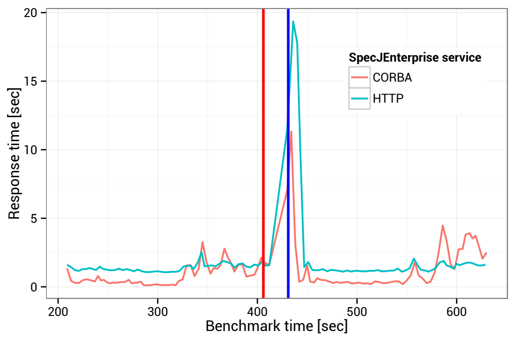
2.4.3 Migration overheads

During our experiments about 1500 migrations were triggered. Here, we want to briefly discuss the resource overhead by migrations, in order to better understand the results of the experiments described in the previous subsections.

The mean migration duration was 28.73 s for 1459 migrations with quartiles 17.98 s, 24.03 s, 31.77 s, and 96.73 s. It follows a log-normal distribution with $\mu = 3.31$ and $\sigma = 0.27$ as shown in Figure 2.3a. Figure 2.3b shows the impact of a migration on the response time of a VM. Migration start and end times are marked by vertical lines. With the start of the migration, response times were slightly increased. A response time spike can be observed towards the



(a) Migration time histogram



(b) Service degradation

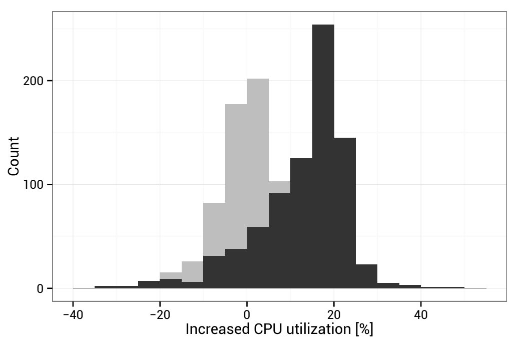
Figure 2.3: VM migration using KVM. 2.3a is a histogram of the migration duration over a couple of hundred migrations. 2.3b shows the response time of a VM that gets migrated.

end of the migration. At this time the VM was handed over from one host to the other. Similar effects were described by [94] and [60].

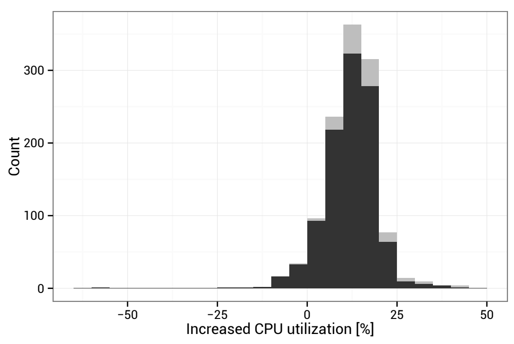
Migration algorithms work by tracking the write operations on memory pages of a VM which consumes additional CPU cycles in the hypervisor [1]. Both dynamic controllers triggered only one migration at a time for each server. For each migration the mean CPU load for 60 s before the migration and during the migration was calculated. Both values were subtracted which provides an estimate for the CPU overhead of a migration.

On the source server an increased CPU load with a mean of 7.88 % and median of 8.06 % was observed. Not all deltas were positive as seen in Figure 2.4a which can be explained by the varying resource demand during the migration on other VMs running on the same server. Only servers with a CPU utilization below 85 % were considered for the histogram. The gray histogram area considers all migrations. In this case, many migrations did not lead to a CPU overhead as utilization cannot increase beyond 100 %. For the target servers the CPU utilization increased by 12.44 % on average. Again, the CPU utilization also decreased in some cases (see Figure 2.4b).

Network utilization is one of the main concerns when using migrations. Similar to today's data centers, all network traffic was handled by a single network



(a) Source server



(b) Target server

Figure 2.4: Histograms on the CPU overheads of migrations on the source- 2.4a and target-server 2.4b. Overhead is calculated as the difference of average CPU during and before a migration. All (gray) and servers with $\leq 85\,\%$ load (black).

interface. Similar to CPU, we calculated the delta of the network throughput before and during migrations. The difference on the source and target server was close to 70 MBps. This indeed illustrates that a separate network for migrations will be necessary if dynamic control is being used.

Network throughput benchmarks report a maximum throughput of 110 MBps for a gigabit connection. This throughput is not achieved as our measurements include some seconds before and after the network transfer phase of a migration. Also, a migration is not a perfect RAM to RAM transfer as the algorithm has to decide on which memory pages to transfer. The 95th percentile of our network throughput measurements during a migration was 105 MBps which is close to the throughput reported in benchmarks. Network overloads were ruled out due to the use of a LACP bond with two gigabit connections where one was dedicated for migrations.

When a VM is handed over from its source to the target server a short service disruption is caused. This was analyzed by [60] for game servers running in VMs. In our experiments the workload generator experienced a short service disruption during the migration which caused some requests to time out. For 83 % of the migrations no errors were observable in the workload generators after and during a migration. The number of errors was 11 for 50 % and 24 for 95 % of all failed migrations. A correlation between source or target server utilization and the number of errors was not noticeable.

2.4.4 Sensitivity analysis

As in any experiment, there are a number of parameter settings which could further impact the result. Especially, for reactive and proactive controllers, the parameters such as the threshold levels were chosen based on preliminary tests. In the following, we provide a sensitivity analysis in order to understand the robustness of the results. We conducted experiments varying the parameters: $T_{underload}$, $T_{Overload}$, K, and M for both dynamic controllers, as well as the control loop interval. MIX2 was chosen because it entails a high variability and is better suited to dynamic controllers. The results are described in Table 2.4.

Changing the threshold settings from $T_{underload} = 40$ and $T_{Overload} = 90$ to $T_{underload} = 20$ only results in a less aggressive controller with a better performance regarding migrations, violations, and average response time. This comes at the cost of an increased average server demand. Setting $T_{underload} = 60$ made the controller more aggressive. Average server demand could be minimized at the price of increased response time, migrations, and violations. The threshold settings certainly depends on the type of workload used. They need to be tuned for each situation and desired service quality. For the experiments we chose a middle way, considering migrations and violations.

We decreased the control loop interval from 300 s to 30 s with negligible impact on the metrics. The average number of servers, violations, and response time are comparable to the results that we found in previous experiments while the number of migrations was slightly increased.

The K and M values describe for how long an overload situations has to prevail until a controller acts upon it. Setting $K = 50, M = 45$ had a slightly positive effect on all metrics. Changing it to $K = 10, M = 8$ yielded a more aggressive controller with more migrations. It triggered up to 60 migrations compared to 50 before without a positive effect on average server demand.

Controller	$\overline{\text{Srv}}$	$\overline{\text{CPU}}$	$\overline{\text{RT}}$	$\lceil\text{RT}\rceil$	$\frac{\overline{\text{O}}}{[\text{sec}]}$	I_{late}	O_{fail}	Mig	SQ
MIX2 + Reactive Controller									
$K = 10, M = 8$	4.42 (0.03)	56.21	564	75110 (14560)	79 (0)	930 (239)	37 (24)	40.67 [24/60]	99.28%
$K = 50, M = 45$	4.03 (0.11)	56.00	536	71797 (15188)	79 (0)	771 (367)	30 (12)	21.33 [16/29]	99.41%
$T_{Underload} = 20$	5.57 (0.73)	42.74	502	49194 (28580)	80 (0)	747 (382)	12 (11)	11.33 [9/15]	99.42%
$T_{Underload} = 60$	3.96 (0.1)	60.06	571	81481 (26900)	79 (0)	1001 (143)	46 (17)	43.33 [28/63]	99.23%
control interval = 30	4.22 (0.06)	59.61	584	72763 (14529)	78 (1)	803 (313)	47 (1)	39.33 [31/48]	99.38%
MIX2 + Proactive Controller									
AR forecast	4.15 (0.25)	56.34	539	56599 (2138)	79 (0)	755 (155)	22 (7)	23.33 [20/29]	99.42%
AR forecast $M = \text{inf}$	3.78 (0.5)	60.40	641	58231 (22600)	78 (1)	1671 (1182)	12 (8)	19.33 [7/29]	98.71%
DES $\alpha = 0.2, \gamma = 0.3$	3.91 (0.65)	59.97	650	57861 (21786)	78 (1)	1516 (1254)	25 (29)	22.33 [7/37]	98.83%
DES $\alpha = 0.5, \gamma = 0.1$	3.85 (0.08)	57.66	533	72645 (30836)	78 (3)	674 (136)	37e3 (64e3)	21.33 [19/24]	99.48%

Table 2.4: Experiments to test the sensitivity of reactive and proactive controller parameters. K, M, T – parameters of the reactive and proactive controllers, $\overline{\text{Srv}}$ – average server demand, $\overline{\text{CPU}}$ [%] – average CPU utilization, $\overline{\text{RT}}$ $[ms]$ – average response time, $\lceil\text{RT}\rceil$ $[ms]$ – maximum response time, $\frac{\overline{\text{O}}}{[\text{sec}]}$ – average operations per second, I_{late} – 3 s intervals with $\overline{\text{RT}} > 3$ s, O_{fail} – failed operations count, Mig – VM migration count, SQ [%] – service quality based on 3 s intervals

In addition we tested different forecast settings for the proactive controller. We used an AR model instead of a double exponential smoothing. The average server demand, migration count, average response time, and max. resp. time roughly were on the same level as for previous experiments. Setting $M = \infty$ calculates the AR forecast with all available utilization measurements. The average server demand improved but it created more response time violations, negatively affecting service quality.

Modifying the $\alpha = 0.2$ and $\gamma = 0.1$ variables of the DES had no significant effect. Increasing $\alpha = 0.5$ yielded similar results then the default configuration. $\gamma = 0.3$ resulted in more violations and slightly increased average response times without an effect on the average server demand.

2.4.5 Simulations for larger scenarios

It is important to understand how the results of our simulations compare to those of experiments, considering the parameter settings and migration overheads learned from experiments. In case simulations will yield comparable results, we wanted to understand, how the performance metrics develop with growing environments regarding more servers and VMs.

Our discrete event simulation framework consists of a workload generator, a controller, and an infrastructure model. Servers and VMs are stored in the model together with their allocation. Each VM was assigned a workload profile. The workload generator iterates over all VMs and updates their current CPU utilization according to their workload profile in 3 s intervals – the same frequency as utilization measurements were received from Sonar during an experiment.

The framework does not reproduce the detailed interactions of web, application, and database server in a VM. It sums the CPU utilization of all VMs to estimate the utilization of a server. Therefore, we do not report response times or operations per second. Instead we count the 3 s intervals with CPU overload, where the accumulated CPU load of a server exceeds 100 %.

The controller was activated every 5 min and triggered migrations according to the server utilization. The same model and controller implementations were used as in the experiments. Migration duration was simulated using a log-normal distribution with the parameters we experienced during experiments (see Section 2.4.3). An additional CPU overhead of 8 % on the source and 13 % on the target server was simulated during migrations. Simulations used the same workload profiles as experiments.

Table 2.5 shows the results of simulations with 6 servers and 18 VMs for workload profiles MIX1-3. For static controllers, simulations and experiments yielded identical VM allocations and have the same server demand. For dynamic controllers, average server demand is not identical but closely matches the one experienced in experiments. Service quality was similar for simulations and experiments. However, this metric should be considered with care as it was calculated differently in simulations.

Simulations triggered more migrations than experiments. Most likely this is due to the differences in workloads used. CPU utilization traces from Sonar typically exhibit a high variance. As K out of M measurements have to be

Controller	$\overline{\text{Srv}}$	$\lfloor\text{Srv}\rfloor$	$\lceil\text{Srv}\rceil$	Mig	SQ
		MIX 1			
Optimization	6.00	6.00	6.00	0.00	100.00
Overbooking	5.00	5.00	5.00	0.00	84.71
Proactive	5.62	4.00	6.00	30.00	99.16
Reactive	5.74	5.00	6.00	34.00	98.99
RoundRobin	6.00	6.00	6.00	0.00	96.88
		MIX 2			
Optimization	4.00	4.00	4.00	0.00	100.00
Overbooking	3.00	3.00	3.00	0.00	90.56
Proactive	3.75	3.00	6.00	40.00	98.23
Reactive	4.22	3.00	5.00	33.00	98.63
RoundRobin	6.00	6.00	6.00	0.00	100.00
		MIX 3			
Optimization	5.00	5.00	5.00	0.00	100.00
Overbooking	4.00	4.00	4.00	0.00	95.31
Proactive	4.69	3.00	6.00	43.00	99.21
Reactive	5.02	4.00	6.00	35.00	98.64
RoundRobin	6.00	6.00	6.00	0.00	98.77

Table 2.5: Simulation results for MIX1-3. $\overline{\text{Srv}}$ – average server demand, $\lceil\text{SD}\rceil$ – maximum server demand, $\lfloor\text{SD}\rfloor$ – minimum server demand, Mig – VM migration count, SQ [%] – service quality based on 3 s intervals

above or below a threshold, a high variance decreases the chances to detect an overload or underload situation and leads to fewer migrations.

The comparison between simulation and experiment show that simulation results need to be interpreted with care, even if the same software infrastructure and parameter estimates are used. While there are differences in the number of servers used, the differences are small. Hence, we use simulations as an estimator to assess how the average server consumption will develop in larger environments.

We examined scenarios with up to 360 VMs and approximately 60 servers. As MIX1-3 only contain 18 workload profiles each, new workload profiles for the simulation were generated. These traces were prepared as described in Section 2.2.2 and labeled as MIXSIM. The simulation results for scenarios with 18, 90, 180, and 360 VMs are shown in Table 2.6. For each treatment three simulations were conducted and their mean value is reported. For each simulation, the set of workload profiles assigned to the VMs was sampled randomly from MIXSIM.

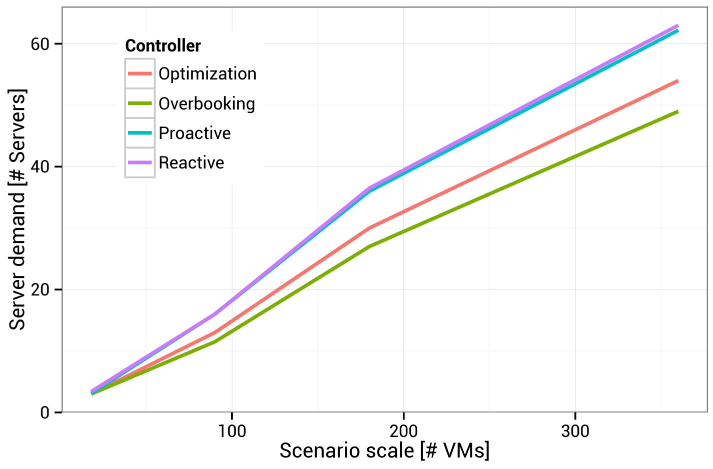


Figure 2.5: Growth of the number of servers required for different numbers of VMs

For the static server allocation problem, computational complexity increases with the number of servers and VMs. Optimizations with six servers are still solvable with workload profiles at a sampling rate of three minutes, while instances with 30 or more servers are only solvable at a sampling rate of 1 h without an optimal solution within 60 min of calculation time, leading to a decreased solution quality. The computational complexity and the empirical hardness of the problem was discussed by [86]. Hence, for larger problem sizes of 60 VMs or more, we computed allocations based on the algorithms introduced by [77]. It leverages SVD to compute near-optimal solutions even for larger problem sizes with several hundred VMs.

Figure 2.5 shows that with an increased number of VMs the number of servers required increases in all controllers, but with a lower gradient for the optimization-based static controllers. Consequently, the advantage of static controllers actually increase with larger numbers of VMs.

Controller	$\overline{\text{Srv}}$	$\lfloor\text{Srv}\rfloor$	$\lceil\text{Srv}\rceil$	Mig	SQ
Tiny (18 VMs)					
Optimization	3.00	3.00	3.00	0.00	100.00
Overbooking	3.00	3.00	3.00	0.00	92.17
Proactive	3.06	3.00	3.10	1.90	99.93
Reactive	3.12	3.00	3.40	2.80	99.85
Small (90 VMs)					
Optimization	13.00	13.00	13.00	0.00	98.84
Overbooking	11.50	11.50	11.50	0.00	86.88
Proactive	15.04	14.50	16.00	33.50	99.52
Reactive	15.13	14.50	16.00	41.50	99.57
Medium (180 VMs)					
Optimization	30.00	30.00	30.00	0.00	99.86
Overbooking	27.00	27.00	27.00	0.00	96.99
Proactive	32.48	30.00	36.00	39.00	99.74
Reactive	32.65	29.50	36.50	62.50	99.64
Large (360 VMs)					
Optimization	54.00	54.00	54.00	0.00	98.91
Overbooking	49.00	49.00	49.00	0.00	87.19
Proactive	59.58	57.00	62.20	82.20	99.79
Reactive	59.80	57.00	63.00	122.60	99.72

Table 2.6: Simulations for MIXSIM with different number of servers and VMs. $\overline{\text{Srv}}$ – average server demand, $\lceil\text{SD}\rceil$ – maximum server demand, $\lfloor\text{SD}\rfloor$ – minimum server demand, Mig – VM migration count, SQ [%] – service quality based on 3 s intervals

2.4.6 Managerial impact

From a managerial point of view, it is interesting to analyze potential energy savings of different allocation controllers. This requires some assumptions about energy costs. Based on the average energy consumption of 10 kW per server rack[3], an average energy consumption of 238 W for a 1U server is assumed. Electricity prices vary depending on the global location and the tariff. In the following we assume a price of 16 $/MWh, such that a server costs 334 $ for electricity which is 40 045 $ for 120 servers over one year.

As shown in Table 2.5, an aggressive static allocation can save up to 50 % compared to a round robin allocation. About 70 servers can be saved which translates into yearly energy cost savings of 23 360 $. In addition, around 70 000 $ could be saved for these servers by considering a 4 years write off

[3]http://www.datacenterdynamics.com/research/energy-demand-2011-12

period and a server cost of 4000 $. Aspects like cooling, emergency generators, rack space and supportive facilities have to be considered as well. A total cost of ownership calculation is out of the scope of this work, however.

2.5 Conclusions

Dynamic controllers are often seen as the next step of capacity management in virtualized data centers promising higher allocation efficiency regarding operation and management. Energy is a significant cost driver in data centers and many IT service managers need to make a decision on static or dynamic allocation controllers. Unfortunately, there is hardly any empirical evidence for the benefits of dynamic ones so far. In this work, we provide the results of an extensive experimental study on a testbed infrastructure. We focus on private cloud environments with a stable set of VMs that need to be hosted on a set of servers. We leverage data from two large European IT service providers to generate realistic workloads in the testbed.

In experiments we found dynamic controllers not to decrease average server demand with typical enterprise workloads. Depending on the configuration and the threshold levels chosen they can lead to a large number of migrations, which negatively impacts application response times and can even lead to network congestion in larger scenarios. Simulations showed that optimization-based static controllers provide better results compared to dynamic ones for large environments as possibilities to leverage workload complementarities increase.

Any experimental study has limitations and so has this. First, a main assumption of the results is the workload, which is characterized in the Appendix E. We have analyzed workloads with high volatility and even added additional noise with robust results. The results do not carry over to applications with workloads that are difficult to predict. For example, order entry systems can experience demand peaks based on marketing campaigns, which are hard to predict from historical workloads. Also, sometimes VMs are set up for testing purposes and are only needed for a short period of time. In such cases, different allocation controllers are required and dynamic controllers clearly have their benefits. Such applications are typically hosted in a separate server cluster, and we leave the analysis of such workloads for future research.

Second, the experimental infrastructure was small and the results for larger environments are based on simulations. While simulations have their limitations, we took great care that the main system characteristics such as migration duration were appropriately modeled. Also, the controller software was exactly the same as the one used in experiments. Overall, we believe that at least the number of servers required by allocation controllers provides a good estimate.

Finally, one can think of alternative ways to implement dynamic controllers. For example, advanced workload prediction techniques [65, 16] could be used. We conjecture, however, that the basic trade-off between migration costs and allocation density gains by dynamic controllers will persist with more sophisticated allocation controllers.

VM migrations are always expensive because they require additional network bandwidth and even can saturate a separate dedicated network. Static controllers with manual or seasonal reallocation only require a fraction of the migration from dynamic controllers. We have seen that 20 % of the migrations cause application errors. Future migration algorithms might address some of these issues but will still lead to some level of overhead and risk as the whole VM memory needs to be transferred over the network physically.

Although the study shows that with a stable set of business applications static controllers with a modest level overbooking would lead to the highest allocation density, we suggest that in everyday operations, a combination of both mechanisms should be put in place where allocations are computed for a longer period of time and exceptional workload peaks are treated by a dynamic controller. However, migrations should be used in exceptional cases only.

Chapter 3

The DSAP dynamic controller

Our objective is to optimize the allocation of VMs to servers so that server demand gets minimized. Static allocations are maintained for a longer period of time until they get recalculated. Especially enterprise data center workloads are predictable as they show a strong seasonality (Chapter 2 and Appendix E). Multiple VMs with complementary workloads are predestined to be allocated on the same server as they do not interfere with each other.

Static controllers cannot deliver very good results if most VMs exhibit a very similar, predictable workload behavior. Dynamic controllers are able to cover with such scenarios. They can reallocate VMs to servers. Migration technology is used to move a VM during operation from one server to another without noticeable service disruption [60]. Off-line dynamic controllers are able to calculate a dynamic allocation that migrates VMs to a small set of servers during periods of low utilization and expands them to a larger set of servers during times of high utilization.

Our work is based on the dynamic server allocation program (DSAP) originally proposed by [9] as an extension of the static server allocation program (SSAP). We contribute simulations as well as experiments on the performance of DSAP based on realistic workload scenarios in a testbed infrastructure that closely resembles the architecture found in private IaaS clouds. Such experiments are expensive in terms of investment costs, implementation effort as well as execution time. Nevertheless, they are necessary as simulations cannot cover the many system invariants like VM migration overheads.

Our findings are that DSAP can increase VM allocation density at the price of system stability and service quality degradation. Frequent reallocations are required to reduce average server demand, leading to a high number of VM migrations. Each migration entails a considerable CPU and memory footprint, potentially leading to additional resource bottlenecks and service quality degradation. Longer reallocation cycles reduce migrations and counteract service quality degradation with a negative effect on average server demand. In summary, benefits in operational efficiency cannot outweigh the costs entailed by many VM migrations.

Scalability of DSAP is very limited as it is based on an NP-hard integer program [9]. In contrast to heuristics it is able to calculate optimal allocations. Our results demonstrate how migrations, server demand, and service quality would perform under optimal conditions. They indicate that VM allocation heuristics should focus on reducing migrations. Establishing optimal VM allocations still becomes relevant if migrations are at a low level.

Section 3.1 covers related work and Section 3.2 describes the DSAP optimization model used to calculate dynamic allocations. Section 3.3 explains the DSAP model implementation for experiments and simulations. Simulation results are discussed in Section 3.4 and Section 3.5 covers our experimental design and results.

Texts in this chapter are based on a previous publication [97].

3.1 Related work

Dynamic controllers actively monitor VM and server utilizations and migrate VMs on demand. Sandpiper [101] migrates if a server's utilization surpasses a certain threshold. Gmach et al. [33] leverage fuzzy logic to implement a similar reactive control approach. In addition an approach similar to SSAPv is used to further enhance the allocation. VMWare DRS [37] load-balances VMs in a cluster based on adaptive thresholds.

pMapper [92] calculates a new allocation over all VMs to estimate a target utilization level for each server. Many migrations would be necessary in order to establish this allocation. An iterative approach is used to trigger migrations so that the desired server target utilization is achieved without considering the calculated VM allocation at all. An additional cost-benefit ratio calculation is used to further reduce migrations.

DSAP [9] is an extension of the SSAP program that calculates multiple independent static allocations instead of just one. Each allocation is valid for a certain period. In contrast to pMapper allocations get realized. [78] propose dynamic server allocation program plus (+) migration overheads (DSAP+) that considers migration overheads. We found migration overheads to be significant but were unable to use DSAP+ due to its computational complexity.

In this study we focus on whether off-line dynamic controllers like DSAP are viable alternatives to on-line dynamic controllers such as Sandpiper, pMapper, or controllers used in Chapter 2. As DSAP generates optimal allocations it provides valuable insights on how to optimize such dynamic controllers. Shrinking and growing the set of active servers in dependence to the actual workload sounds promising. Still, it is unclear if the overhead entailed by VM migrations nullifies or exceeds the benefits of a dynamic allocation.

3.2 Dynamic server allocation program

The original DSAP integer program was proposed by [9]. We leverage it to calculate dynamic allocations that are evaluated in simulations and experiments.

For reasons of completeness we provide the formal definition [9] of DSAP in Equation 3.1. Suppose we are given a set of servers $i \in I$ and VMs $j \in J$. A server's size is denoted by s_i describing its resource capacity, e.g. CPU units or available memory. The total planning horizon is divided into τ discrete equidistant periods $t \in \tau$. $y_{it} \in \{0,1\}$ indicates whether a server i is active in period t. u_{jt} describes the utilization of VM j in period t. The allocation matrix x_{ijt} of period t indicates whether VM j is assigned to server i. Overall server demand gets minimized by the objective function.

$$
\begin{aligned}
&\min \sum_{t=1}^{\tau}\sum_{i=1}^{I} y_{it} && \\
&\text{s.t.} && \\
&\sum_{i=1}^{I} x_{ijt} \qquad = 1, && \forall j \in J, \forall t \in \tau \\
&\sum_{j=1}^{J} u_{jt} x_{ijt} \qquad \le s_i y_{it}, && \forall i \in I, \forall t \in \tau \\
&y_{it}, x_{ijt} \in \{0,1\}, && \forall i \in I, \forall j \in J, \forall t \in \tau
\end{aligned}
\tag{3.1}
$$

The program covers only one critical resources, e.g. CPU or memory. Multiple resources $k \in K$ can be considered by altering the second set of constraints as shown in Equation 3.2.

$$
\sum_{j=1}^{J} u_{jkt} x_{ijt} \le s_{ik} y_{it}, \quad \forall i \in I, \forall k \in K, \forall t \in \tau \tag{3.2}
$$

A dedicated allocation x_{ijt} is calculated for each period t. VM migrations transition the infrastructure from its current allocation x_{ijt} to the subsequent one $x_{ij(t+1)}$. Each migration puts additional load on the servers involved as shown by [1]. Our infrastructure is based on KVM instead of XEN. Migrations on average increase the CPU load on the source and target server by approximately 8 % to 13 % as shown in Chapter 2. Data center operators might put a limit Z on the migration count by adding two additional constraints [9] as shown in Equation 3.3.

$$
\begin{aligned}
x_{ijt} - x_{ij(t-1)} &\leq z_{ijt} \quad \forall i \in I, \forall j \in J, \forall t \in \{2, \dots, \tau\} \\
x_{ij(t-1)} - x_{ijt} &\leq z_{ijt} \quad \forall i \in I, \forall j \in J, \forall t \in \{2, \dots, \tau\} \\
\sum_{i=1}^{I} \sum_{j=1}^{J} \sum_{t=2}^{\tau} z_{ijt} &\leq 2Z \\
z_{ijt} \in \{0,1\}, & \qquad\quad \forall i \in I, \forall j \in J, \forall t \in T
\end{aligned}
\tag{3.3}
$$

They increase computational complexity by adding a large number of binary decision variables z_{ijt}. We were not always able to solve the model optimally for 6 servers, 18 VMs, and 6 periods within one hour using an Intel Core i7 870 CPU and a Gurobi solver. Calculation time was limit to 60 min for simulations and experiments, possibly yielding suboptimal solutions.

Both constraints might have a negative effect on allocation density. Migrations are caused by differences between subsequent allocations x_{ijt} and $x_{ij(t+1)}$. It might be possible that both constraints lead to similar subsequent allocations without a negative effect on overall allocation density. On the other hand, they might have a strong negative effect on allocation density if certain highly efficient allocations are getting ruled out by these constraints.

3.3 Experimental setup

Our software framework provides a number of application programming interfaces (APIs) that allow to control the testbed infrastructure, receive resource load measurements on all servers and VMs, and to query workload profiles u_{jt}. An extended description of the testbed infrastructure is provided in Appendix A.

The DSAP controller execution starts with an initialization phase as shown in Figure 3.1. It solves the DSAP model for a given set of workload profiles u_{jt}. VMs are allocated to the servers according to the first allocation x_{ij1} using migrations. Each period has a fixed duration of $^{simtime}/_{\tau}$ where *simtime* is the duration of one experiment or simulation in hours. After each period

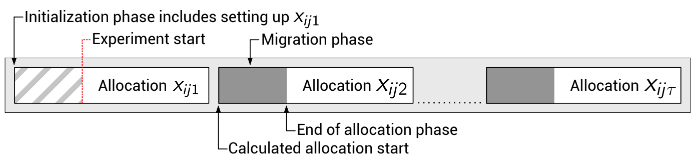


Figure 3.1: Migration phase between two stable allocations

a new allocation gets activated. VM migrations are triggered to adjust the infrastructure accordingly.

The controller determines all changes between the current and next allocation to create a list of necessary migrations. Launching all migrations at the same time would possibly lead to multiple migrations on a single server in parallel. To avoid server overloads only one migration is allowed at a time for each server.

Migrations are scheduled by a migration queue. For all migrations it is checked whether they can be executed now without intersecting with active migrations in terms of using the same server or VM. This process is repeated each time a migration finishes as long as the migration queue contains entries.

Migrations are executed during a migration-phase at the beginning of each period (see Figure 3.1). Its duration is determined by the speed and number of migrations. Likelihood for service disruption is increased as DSAP does not consider migration overheads in terms of resource consumption or duration.

The intermediate allocation state during a migration-phase is not considered by the DSAP either. This might cause overload situations as servers might be oversubscribed. Worse, the migration-phase cannot be expected to be short compared to the duration of a period. For example, if 10 migrations are executed in sequence with an average migration time of 30 s (see Chapter 2) the migration-phase takes approximately 5 min or 8 % of a 60 min period length.

For simulations and experiments we used a set of 451 CPU and memory utilization or application request throughput traces from two European IT service providers. Descriptive statistics on this data set are provided by [86] and in

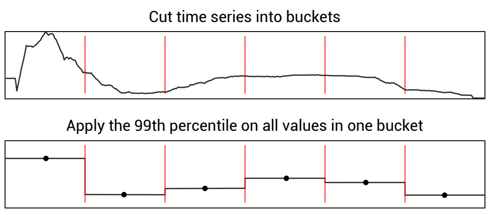

Cut time series into buckets
Apply the 99th percentile on all values in one bucket

Figure 3.2: Profile downsampling for the DSAP parametrization

Appendix E. We focused on the CPU utilization traces that show strong seasonality for days and weeks. We extracted a workload profile for each trace that represents the typical daily workload of the corresponding server following the same approach used in Chapter 2 and Appendix E.

This workload profiles were used to parametrize the DSAP, drive simulations and to generate workload for the testbed infrastructure. As in Chapter 2, the first set contains profiles with low volatility, the second one holds volatile profiles. A third mix MIX3 combines MIX1 and MIX2 by randomly sampling 9 traces from each without replacement.

A workload profile is stored as a time series with a sampling rate of 3 min over a duration of 6 h which is the duration of one simulation and experiment. DSAP requires a time series of τ data points. The workload profile was downsampled by splitting it into τ parts and taking the 99th percentile of the data points in each part (see Figure 3.2). We conducted simulations with other percentiles as shown in Section 3.4 and found the 99th percentile to be a good choice in order to maintain a high service quality.

The infrastructure used to conduct simulations and experiments is described in Appendix A. The DSAP controller implementation is independent to the operation mode, either simulation or experiment.

3.4 Simulation results

We conducted a number of simulations to asses how DSAP will work for different input parameters. Simulations precede experiments as these are costly in

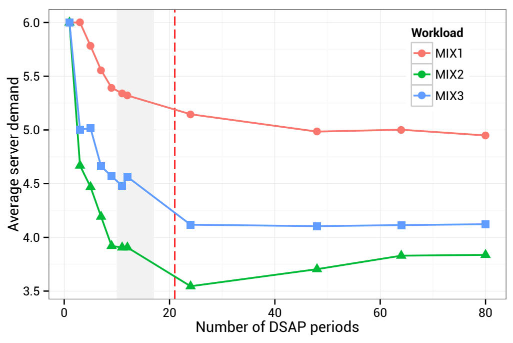


Figure 3.3: DSAP gets more efficient if the number of periods is increased

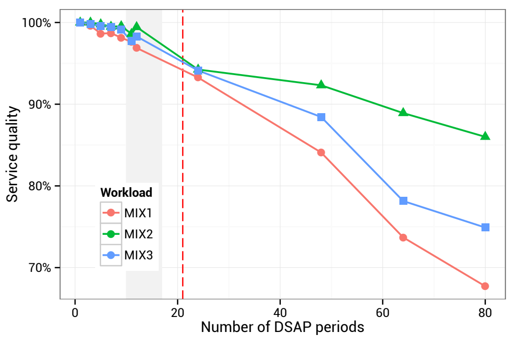


Figure 3.4: Service quality degrades if the number of periods is increased

terms of time.

Simulations were conducted for all DSAP configurations and workload mixes MIX1-3 in an environment of 6 servers and 18 VMs. A server had 15 GB of memory available to the VMs, each consuming 2 GB. VM CPU workload profile values vary in a range of $[0, 100]$. Maximum CPU capacity of a server was set to $s_i = \{200, 230\}$. A value of $s_i = 200$ resembles the testbed's hardware setup where two VMs can fully utilize one server, a setting of $s_i = 230$ oversubscribes servers on purpose.

τ was varied between 1 and 60 periods which affects the number of allocations calculated by the DSAP and in turn the number of migrations. Setting $\tau = 1$ effectively reduces DSAP to an SSAP model [9], a simplified version of SSAPv that does not consider workload profiles. By varying the number of periods we wanted to see how the amount of required servers develops. Results are shown in Figure 3.3. For all workloads server demand improves if the number of periods is increased to 20, equivalent to a period length of 18 min. Shorter period lengths do not indicate additional efficiency gains and might even have a negative effect as seen for MIX2 in Figure 3.3.

Figure 3.4 shows that the service quality decreases if more periods are used. The intention of DSAP is to maintain a high service quality while minimizing average server demand. For all workloads the service quality falls off at around 17 periods, a 21 min period length.

With an increasing number of periods and allocations the number of migrations also increases. Each migration comprises additional overhead. At the same time periods get shorter which puts more weight into the migration-phase (see Figure 3.1). Ultimately, the allocation phase length will be equal to the period length. An optimal allocation will not be reached. We conclude that a high number of migrations and an increased allocation phase duration seem to cancel out the benefits of dynamic controllers at some point.

The simulation results indicate that 10 to 17 periods deliver a high efficiency in terms of average server demand and service quality for this particular scenario. For more than 17 periods service quality falls off to an sub-optimal value below 99 %.

In a second set of simulations we tested sensitivity regarding the CPU capacity and the bucket-percentile parameters of the DSAP. Previous experiments with the SSAPv [86] model conducted in Chapter 2 indicated that overbooking often results in a slight but acceptable service quality degradation while at the same time average server demand can be reduced. We wanted to see if this results carry over to DSAP.

Factor bucket percentile was varied between $\{90, 99\}$ and factor CPU capacity s_i between $\{200, 230\}$, giving 2^2 configurations. Each one was simulated for workload mixes MIX1-3. Results are shown in Table 3.1. Controllers are named by an approach similar to the Kendall notation with four variables separated by a slash. The first element declares the bucket percentile, the second one is CPU capacity s_i, number of periods τ is described by the third one, and a potential migration limit Z is stated in the fourth element.

Most of the performance metrics reported are self explanatory, except service quality. It is calculated based on a violation counter, which is incremented each time the driver updates a server utilization that exceeds its capacity. For example, three overloaded servers would account for three violations for one 3 s simulation interval. Similarly, a slot counter is incremented for each simulation interval and for each non-empty server. If five servers are hosting VMs this would account for five increments in one simulation interval. Both values are

Configuration	$\overline{\text{Srv}}$	$\lfloor\text{Srv}\rfloor$	$\lceil\text{Srv}\rceil$	Mig	SQ	$\overline{\text{Srv}}$	$\lfloor\text{Srv}\rfloor$	$\lceil\text{Srv}\rceil$	Mig	SQ	$\overline{\text{Srv}}$	$\lfloor\text{Srv}\rfloor$	$\lceil\text{Srv}\rceil$	Mig	SQ
			MIX1					**MIX2**					**MIX3**		
[90/200/6/∞]	5.19	5	6	76	97.04	3.37	3	5	72	98.42	4.38	4	6	79	97.07
[90/230/6/∞]	4.84	4	6	70	73.40	3.41	3	5	77	97.55	3.86	3	6	70	92.20
[99/200/6/∞]	5.82	5	6	80	98.93	4.38	4	6	73	99.82	4.67	4	6	79	99.33
[99/200/9/∞]	5.39	5	6	113	98.14	3.92	3	6	113	99.55	4.57	3	6	125	99.11
[99/230/6/∞]	5.02	5	6	76	80.87	3.68	3	6	72	97.98	4.37	4	6	70	96.50
[99/230/9/∞]	4.79	4	6	125	83.21	3.57	3	6	121	99.06	3.82	3	6	116	93.90
[99/200/6/30]	5.82	5	6	29	99.43	4.38	4	5	27	99.97	4.66	4	5	21	99.45
[99/230/6/30]	5.00	5	5	20	87.86	3.67	3	6	30	100.00	4.37	4	5	22	98.16
Proactive	5.62	4	6	30	99.16	3.75	3	6	40	98.23	4.69	3	6	43	99.21
Reactive	5.74	5	6	34	98.99	4.22	3	5	33	98.63	5.02	4	6	35	98.64
Optimization	6.00	6	6	0	100.00	4.00	4	4	0	100.00	5.00	5	5	0	100.00
RoundRobin	6.00	6	6	0	96.88	6.00	6	6	0	100.00	6.00	6	6	0	98.77

Table 3.1: Simulation results comparing the DSAP optimization model with different configurations. Results on controllers: proactive, reactive, optimization, and round robin stem from Chapter 2. Controller – [Bucket percentile, CPU capacity s_i, bucket count τ, migration limit Z], $\overline{\text{Srv}}$ – average server demand, $\lceil\text{SD}\rceil$ – maximum server demand, $\lfloor\text{SD}\rfloor$ – minimum server demand, Mig – VM migration count, SQ [%] – service quality based on 3 s intervals

used to calculate the service quality. It indicates how many simulation intervals were affected by a resource violation.

Independent to the workload mix, varying both factors had only negligible effects on the migration count. Configuration [90/200/6/∞] showed to be aggressive and decreased average server demand sacrificing service quality. Generally, service quality goes up as average server demand increases. Overall [99/200/6/30] without overbooking delivered the best service quality but also had the highest server demand.

For comparison we report simulation results for alternative controllers in Table 3.1 that includes results from Chapter 2. A round robin controller estimates the number of required servers based on VM dimensions. It assigns VMs to servers in a round robin approach during the initialization phase. The optimization controller represents the solution achieved by the SSAPv program [86]. By default optimization is parametrized with a CPU capacity $s_i = 200$ without overbooking. The reactive controller closely monitors the infrastructure and triggers migrations based on threshold violations similar to [101]. For

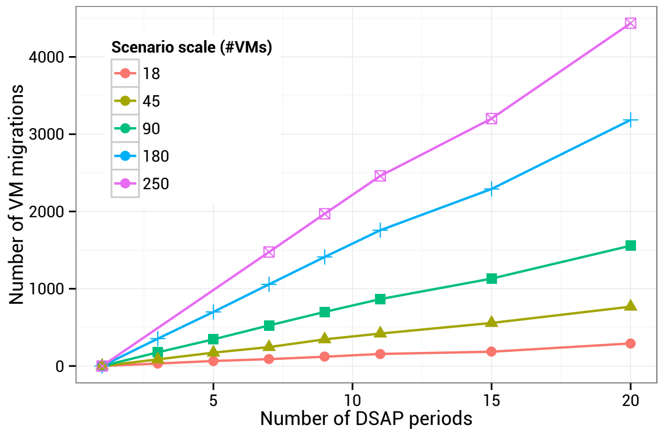


Figure 3.5: Relationship between the number of VM migrations, number of DSAP periods, and the number of VMs

proactive, double exponential smoothing (DES) is used to get a point forecast on the current CPU utilization.

Considering all workload mixes, optimization required slightly more servers then dynamic controllers while delivering a good service quality without triggering any migrations. Round robin used most servers but still was not able to deliver a service quality above 99 % consistently. Reactive delivered a good service quality independent to the workload mix and used less servers than optimization. DSAP required approximately two times as many migrations with 6 periods and three times as many with 9 periods as the reactive controller. Its configurations [99/200/6/∞] and [99/230/6/∞] where the only ones achieving a service quality and server demand on par with the reactive controller.

Migrations always entail the risk of service disruption as they might fail and put additional stress on the network infrastructure. Static controllers like round robin and optimization do not trigger migrations at all. Despite the high number of migrations, DSAP was able to achieve a comparable high service quality.

In a next step larger scenarios were considered. Simulations were parametrized with a CPU capacity of 200, a bucket-percentile of 99, a varying number of periods $\tau = [3, 20]$, and different amount of VMs between $[18, 250]$. Results in Figure 3.5 suggest a linear relationship between the number of periods τ, number of VMs, and migration count.

In order to reduce migrations we conducted simulations with two additional constraints as explained in Equation 3.3. Results are shown in Table 3.1. Due to problem complexity we were unable to solve some instances within one hour. Especially the large number of slack variables z_{ijt} increases computational complexity. Therefore, the best solution found after 60 min was chosen. A migration limit of 30 was set as a similar amount was required by the reactive controller.

Independent of the workload mix comparing previous DSAP simulations with migration-constrained ones does not indicate an increase in server demand. The number of migrations was cut significantly, even below a threshold of 30. Especially configuration [99/200/6/30] without overbooking delivered good results. All metrics are mostly competitive to the reactive controller. Server demand was sometimes increased while service quality was always good. Configuration [99/230/6/30] overbooked servers which decreased average server demand. However, service quality fell far below the desired level of 99 %.

3.5 Experimental results

We wanted to see if simulation results carry over to a real-world environments. For this we executed the same DSAP controller used for simulations in a testbed infrastructure. An extensive description of the infrastructure is provided in Appendix A. Prior to each experiment the infrastructure was reconfigured to match the experimental prerequisites. This includes migrating and restarting VMs in accordance to the initial allocation, resetting the MySQL databases and restarting the Glassfish servers on all VMs.

A single experiment took 6 h. Because of the huge amount of time required to conduct experiments it was impossible to evaluate a large number of treatments. Instead, simulation results were used to come up with promising DSAP settings. The number of periods was set to $\tau = \{6, 10\}$ and server CPU capacity was set to $s_i = \{200, 230\}$. Experimental results are shown in Table 3.2 including alternative static and dynamic controllers that were evaluated in Chapter 2. Again, controllers are named by an approach similar to the Kendall notation with three variables separated by a slash. The first element declares the number of periods τ, CPU capacity s_i is described by the second element, and a migration limit Z is stated in the third element.

Controller	$\overline{\text{Srv}}$	$\overline{\text{RT}}$	$\lceil\text{RT}\rceil$	$\frac{\overline{\text{O}}}{[\text{sec}]}$	I_{late}	O_{fail}	Mig	SQ
				MIX 1				
[6/200/∞]	5.83 (0)	934.00	86907 (4727.72)	139.45 (0)	5474 (25)	84 (6)	79 [79/79]	95.78%
[6/230/∞]	5.02 (0)	1309.00	100041 (15354.12)	133.09 (0)	10652 (1759)	104 (63)	75 [75/75]	91.78%
[10/200/∞]	5.26 (0)	1262.00	235511.5 (174438.29)	133.39 (0)	9264 (29)	152 (68)	135 [135/135]	92.85%
[6/200/30]	5.82 (0)	873.00	87264 (5484.32)	141.05 (0)	3976 (770)	72 (17)	26.5 [25/28]	96.93%
[6/200/30]*	5.82 (0)	1345.00	107890.5 (10154.76)	149.67 (0)	13363 (3544)	51 (31)	27.5 [25/30]	89.69%
Proactive	5.95 (0.07)	566	42012 (5958)	147 (2)	1990 (1609)	15 (5)	10.33 [9/12]	98.46%
Reactive	6 (0)	392	21501 (9386)	150 (1)	279 (25)	14 (1)	0.33 [0/1]	99.78%
Optimization	6 (0)	330	17621 (3777)	151 (0)	137 (26)	8 (2)	0 [0/0]	99.89%
				MIX 2				
[6/200/∞]	4.39 (0)	649.00	96395 (18367.81)	77.57 (0)	1109 (11)	65 (7)	72 [72/72]	99.14%
[6/230/∞]	3.73 (0)	909.00	128203.5 (59140.29)	74.76 (0)	3542 (331)	126 (88)	74.5 [72/77]	97.27%
[10/200/∞]	3.74 (0)	1000.00	457192 (472442.08)	73.15 (0)	5082 (161)	242 (18)	139 [139/139]	96.08%
[6/200/30]	4.35 (0)	667.00	97064 (19241.79)	77.95 (0)	1342 (21)	20 (12)	25.5 [21/30]	98.96%
[6/200/30]*	4.36 (0)	977.00	96113 (17916.67)	94.06 (0)	4512 (2655)	40 (16)	24.5 [24/25]	96.52%
Proactive	3.93 (0.2)	535	65337 (22243)	79 (0)	777 (184)	22 (17)	23 [16/34]	99.4%
Reactive	4.34 (0.18)	547	71153 (23498)	79 (1)	842 (359)	28 (23)	26.4 [18/36]	99.35%
Optimization	4 (0)	467	16875 (7071)	80 (1)	637 (156)	6 (4)	0 [0/0]	99.51%
				MIX 3				
[6/200/∞]	4.7 (0)	735.00	113943.5 (3796.46)	101.26 (0)	1854 (449)	11047 (15443)	70 [70/70]	98.57%
[6/230/∞]	4.39 (0)	1149.00	76163 (12213.15)	96.54 (0)	6424 (303)	70 (13)	76 [76/76]	95.04%
[10/200/∞]	4.49 (0.01)	990.00	112298.5 (4674.68)	91.52 (0)	4865 (751)	150744 (213055)	135.5 [133/138]	96.25%
[6/200/30]	4.68 (0.01)	710.00	83988.5 (614.48)	102.1 (0)	1874 (179)	38 (25)	28.5 [28/29]	98.55%
[6/200/30]*	4.68 (0.01)	1386.00	89875 (8786.51)	113.91 (0)	11378 (4182)	72 (64)	29.5 [29/30]	91.22%
Proactive	4.76 (0.16)	475	54636 (215)	106 (0)	545 (93)	12 (4)	14.33 [10/17]	99.58%
Reactive	4.85 (0.12)	505	59651 (9129)	105 (1)	635 (158)	19 (11)	17 [17/17]	99.51%
Optimization	5 (0)	347	11222 (1171)	107 (0)	73 (6)	8 (2)	0 [0/0]	99.94%

Table 3.2: Experimental results on DSAP. Results on controllers: proactive, reactive, and optimization stem from Chapter 2. Controller – [Periods τ / CPU capacity s_i / Migration limit Z], $\overline{\text{Srv}}$ – average server demand, $\overline{\text{RT}}$ [ms] – average response time, $\lceil\text{RT}\rceil$ [ms] – maximum response time, $\frac{\overline{\text{O}}}{[\text{sec}]}$ – average operations per second, I_{late} – 3 s intervals with $\overline{\text{RT}} > 3$ s, O_{fail} – failed operations count, Mig – VM migration count, SQ [%] – service quality based on 3 s intervals

For each treatment two experiment replications were conducted. Values in parentheses describe the variance and values in squared brackets describe the max and min values. Most performance metrics are comparable to the ones reported for simulations except for service quality. For each VM the Rain workload generator determines the average response time over consecutive 3 s intervals and reports them. A service level agreement (SLA) failure is detected if the average response time of such a 3 s interval exceeds an arbitrarily chosen threshold of 3 s.

Overall DSAP [6/200/∞] delivered the best service quality independent to the workload mix, equally good as the reactive controller. It is the only DSAP configuration that consistently achieved a service quality close to or above 99 %. While delivering similar server savings as the reactive one, migrations and average response time were significantly increased.

Looking at more aggressive DSAP configurations with overbooking or increased number of periods reveals a positive effect on server demand with an average CPU utilization above 70 %. However, service quality decreased as fewer servers were used, confirming simulation results. Operation throughput also decreased, indicating an overloaded infrastructure.

DSAP [6/230/∞] and [6/200/∞] configurations triggered between 70 and 80 migrations over a period of 6 h and 18 VMs. On average a VM gets migrated about 4 times during the experiment. DSAP [10/200/∞] with a period duration of 36 min even triggered 7.5 migrations per VM over 6 hs. Assuming a 2 GB network transfer for each migration results in approximately 240 GB transfer volume over 6 h and 6 servers. This highlights the need for dedicated migration network infrastructure and mechanisms that counteract migrations.

In an additional experiment [6/200/30] we limited migrations to a count of 30. Comparing results with the same configuration without that limitation [6/200/∞] does not indicate any efficiency loss in server demand. Nor did service quality decrease or average response time increase. In most cases even less than 30 migration were triggered. Despite limiting migrations, the reactive controller outperforms DSAP in terms of service quality at a comparable server demand. It triggered even less migrations as predicted by simulations. In summary, limiting migrations seems to be effective as it does not degrade overall allocation density, reduces load on the network infrastructure, and reduces the risk of failing migrations and service disruptions.

Experimental and simulation results on server demand are comparable as calculations are parametrized and calculated equally for simulations and experiments. Compared to static controllers, dynamic ones achieve an additional saving at the cost of migrations. However, savings heavily depend on the workload mix. For MIX1-3 DSAP could not outperform the reactive one and delivers equally good results at best.

All experiments were conducted in a highly deterministic environment. The actual workload experienced by VMs during an experiment closely resembled workload data used by the DSAP optimization program in advance. If the actual workload deviates from the one used by the calculations, the DSAP performance even gets worse. This is shown by the DSAP [6/200/30]* experiment. The workload data of MIX1-3 was used by the calculations but the experiments used strongly modified workload profiles similar to the ones described in Chapter 2. Additional workload peaks were modeled, noise was added, and the workload was shifted randomly between 0 and 30 minutes.

Comparing the experimental results with simulations especially shows differences in the achieved service quality. Simulations still are a viable tool to predict the amount of required servers and the relative number of migrations. Predicting the migration overhead is difficult in simulations. Models that predict the migration duration and impact based on a memory page dirtying rate are not viable as most simulation frameworks do not model the hardware in such a great detail. This example shows the importance of experiments to verify stimulative findings for such controllers.

3.6 Conclusions

In this work we evaluated the DSAP integer program that re-allocates VMs to servers periodically in order to improve allocation density. Its results were compared against static and dynamic controllers that migrated VMs between servers depending on their actual resource utilization.

Simulations on DSAP indicate that frequent VM migrations are necessary to achieve a high overall allocation density. An infrastructure reallocation needs to get triggered every 20 min to achieve server demand savings at an acceptable service quality. Further increasing reallocation frequency does not provide additional savings and even has a negative effect on service quality as much more migrations get triggered.

Experiments in a testbed infrastructure showed that simulation models are quiet accurate with respect to the server demand and migration count but provide only rough estimates on service quality. Over all workload scenarios considered, the reactive controller delivered equally good or better results than DSAP considering core metrics: average server demand, service quality, and migration count.

DSAP triggered between 2 and 3 times more migrations than a reactive controller. Limiting migrations by two additional model constraints leads to a slightly increased server demand without any other negative effects.

In summary, dynamic controllers like DSAP or the reactive controller can achieve an average server demand comparable to static controllers at a reasonable high service quality. However, techniques are required to counteract VM migrations as a main factor for service quality degradation.

Chapter 4

Extending DSAP with migration overheads

For static controllers mathematical problem formulations have been proposed as discussed in Chapter 2 and 3. Static allocations are calculated for an extended period that often covers multiple durations with different utilization characteristics such as holidays, weekends, and night time [70, 86, 76]. Leveraging this information might provide further efficiency gains while allocating VMs.

In Chapter 3 VM migrations were leveraged to implement an off-line dynamic controller that allocates VMs on a short term basis. Short reallocation cycles can reduce average server demand effectively, but lead to a large number of migrations and service quality degradation.

The same approach can also be used to switch between different allocations that are valid for an extended period. For example, one allocation could be used during nighttime and a second one during working hours. Gmach et al. [34], amongst others, proposes a cyclic re-computation of VM assignments after periods of some hours in order to further reduce the required server demand and align to utilization changes. Longer cycles such as days, weeks and months have been proposed by Rolia et al. [71] or Bichler et al. [10].

Each migration entails additional resource overheads on the source server as well as on the target server, especially regarding CPU and memory capacity as shown in Chapter 2, 3, and by [38, 1]. Depending on the frequency an application writes data into the main memory, the CPU overhead varies between 5 % and 30 % of the application's current CPU utilization [1].

The proposed approaches have in common that migration overhead is considered only in an indirect fashion by either defining threshold values with large safety buffers or in restricting the amount of migrations to be allowed in a certain time frame. Assuming deterministic or at least a well predictable VM utilization behavior, migration overheads as well as migration durations can be estimated with sufficient accuracy [1].

VM migrations must be scheduled in a controlled fashion in order to avoid server overloads during a migration. In particular, migrations are only allowed to be triggered if there are enough resources available on the migration source and target servers.

In this chapter we introduce an extension to the DSAP model called DSAP+. It takes migration overheads into account while minimizing average server demand. In contrast to other approaches, it does not rely on predefined thresholds as the reactive controller used in Chapter 2 or migration limits as used in Chapter 3.

Texts in this chapter are based on a previous publication [78].

4.1 Model formulation

We extended the DSAP integer program [79] formulation that was already used in Chapter 3 to cover migration overheads. For each VM it assumes a workload profile with a utilization value for each period. For example, a workload profile of 24 values describes one day, assuming a period duration of 1 h. A dedicated static allocation is calculated for each period.

Consecutive allocations are established by triggering migrations. Each migration generates additional CPU load on the migration source and target server.

The proposed model takes migration overheads into account while calculating consecutive allocations. It is assumed that all migrations are triggered in parallel at the beginning of a period (see Figure 3.1).

A formal model description is shown in Equation 4.1. Suppose we are given a set of servers $i \in I$ and VMs $j \in J$. A server's size is denoted by s_i describing its resource capacity, e.g. CPU units or available memory. The total planning horizon is divided into τ discrete equidistant periods $t \in \tau$. $y_{it} \in \{0, 1\}$ tells whether a server i is active in period t. u_{jt} describes the utilization of VM j in period t. The allocation matrix x_{ijt} of period t indicates whether VM j is assigned to server i.

VM migrations from a source to a target server are indicated by slack variables z^-_{ijt} for outgoing and z^+_{ijt} for incoming migrations. Each migration entails a certain resource overhead m^- for outgoing and m^+ for incoming ones.

The objective function minimizes the overall server demand. For simplicity, the model formulation shown in Equation 4.1 assumes a single limiting resource, e.g. CPU or memory. An extension to cover multiple resources is possible as shown for the underlying DSAP program in Equation 3.2.

$$
\begin{aligned}
&\min \sum_{t=1}^{\tau} \sum_{i=1}^{I} y_{it} \\
&\text{s.t.} \\
&\sum_{i=1}^{I} x_{ijt} && = 1, && \forall j \in J, \forall t \in \tau \\
&\sum_{j=1}^{J} u_{jt} x_{ijt} + m^- z^-_{ijt} + m^+ z^+_{ijt} && \leq s_i y_{it}, && \forall i \in I, \forall t \in \tau \\
&- x_{ij(t+1)} + x_{ijt} - z^-_{ijt} && \leq 0 && \forall i \in I, \forall j \in J, \forall t \in \tau \\
&x_{ij(t+1)} - x_{ijt} - z^+_{ijt} && \leq 0 && \forall i \in I, \forall j \in J, \forall t \in \tau \\
&y_{it}, x_{ijt}, z^-_{ijt}, z^+_{ijt} \in \{0, 1\} && && \forall i \in I, \forall j \in J, \forall t \in \tau
\end{aligned}
\tag{4.1}
$$

The first set of constraints ensures that each VM is allocated to one of the servers at any point in time. The second set of constraints ensures that the aggregated resource demand of multiple VMs does not exceed a server's capacity during all periods. The term on the left hand side of the sum describes the resource demand due to the operation of VMs. The term on the right hand side determines migration-related overheads.

With the third and fourth constraint type the slack variables z^-_{ijt} and z^+_{ijt} used in the second set of constraints are set to one if a VM is migrated away or towards a server. In the next section we will evaluate this model using monitoring data on CPU utilization from two large European IT service providers (see Appendix E).

4.2 Simulation setup

We study the solution quality in terms of average server demand compared to a solution determined by the SSAPv model proposed by [79]. A static controller is used as a benchmark as all approaches that allow for VM migrations found in the literature do not guarantee overload avoidance during migrations even in case of predictable VM resource utilization. CPU was considered as the single limiting resource. For each treatment 10 simulations with different sets of VM workload profiles were calculated.

The impact of migration overheads on the allocation efficiency was evaluated as a first factor. CPU overheads for (source server, target server) were varied between three levels: (20 %, 10 %), (30 %, 20 %), and (40 %, 30 %). In practice, migration overheads depend on the application and it's workload characteristics.

As a second factor, we analyzed the impact of the servers CPU capacity. Small servers (S) could host 3 VMs on average, medium servers (M) were able to host 5 VMs on average, and large servers (L) could host 8 VMs on average.

In summary, we conducted 180 simulations with 3 different CPU migration overhead settings, 3 different server sizes, 10 workload profiles, and two controller implementations (SSAPv and DSAP+).

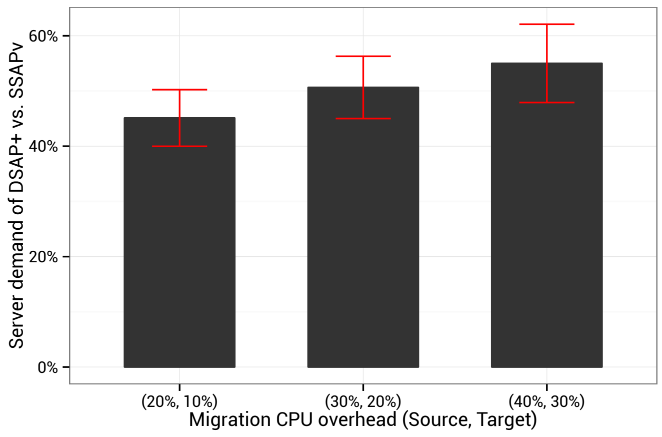


Figure 4.1: Server savings vs. VM migration overhead

4.3 Simulation results

Aggregated results are shown in Figure 4.1. The dark bars show the average server demand of DSAP+ compared to SSAPv over 10 simulations for 3 different migration overheads. For example, the height of 45 % of the outer-left bar indicates a reduction of 55 % on the average server demand when applying DSAP+ model in a scenario with 20 % on the migration source and 10 % overhead on the migration target server. The red error bars indicate the standard deviation of the average server demand savings over all 10 simulations.

Overall, DSAP+ resulted in average server demand reduction of around 55 % compared to SSAPv with a static allocation if migration overheads were low. 49 % savings were achieved with medium migration overhead, and 45 % with a high migration overhead. We compared these results pairwise and found all differences to be significant at a 1 % level with a Wilcoxon signed rank test. That is mainly because the standard deviation of the average server demand savings was varying between 5 % and 7 %.

The positive correlation between migration overhead and average server demand savings results from the fact that with decreasing overhead, migrations become cheaper. An overhead of zero means that VMs can be moved around

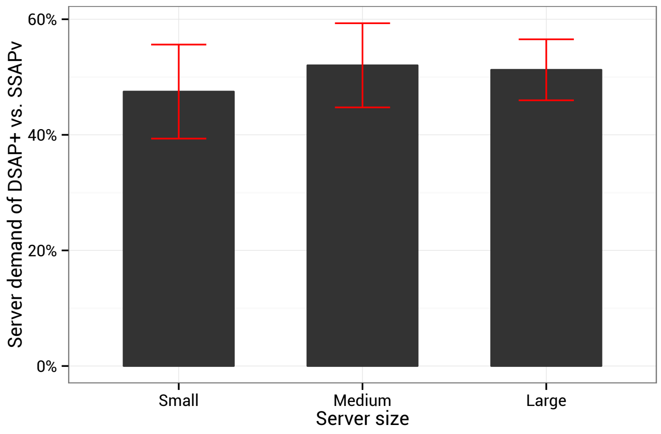


Figure 4.2: Server savings vs. server size

without restrictions in order to evacuate servers whenever possible. Contrary, a very large overhead would not permit migrations at all.

Accordingly, the average number of migrations increased with a decreasing migration overhead. In small server scenarios with high migration overheads, only 73 migrations were triggered, while 84 with medium, and over 100 migrations with small migration overheads. Still, even for relatively large overheads assumed, server demand could be lowered by almost 45 % by DSAP.

Results on the impact of server size on average server savings are depicted in Figure 4.2 for *S*, *M*, and *L* servers.

Average server demand savings vary only slightly between 49 % and 51 % considering different server sizes. A Wilcoxon signed rank test did not find significant differences. We conclude that the cost savings achieved by the DSAP+ model are around 50 % considering different sets of CPU utilization time series and migration overheads.

4.4 Conclusions

We introduced the DSAP+ model as an extension of the DSAP model that takes VM migration overheads into account. In literature some work on reallocating VMs has been discussed, but none considers migration-related overheads directly in a mathematical model formulation. Deterministic simulations, leveraging real-world utilization time series of a data center indicate average server demand savings of around 50 % compared to a static allocation calculated by the SSAPv program.

DSAP+ is meant for data centers that experience seasonal workloads over an extended period, such as weekends where resource utilization is typically low. A majority of servers we analyzed in Appendix E exhibit such predictable, seasonal workload on a weekly basis. In such cases, migration overheads can be predicted with sufficient accuracy. This data allows a parametrization of the DSAP+ model in real-world scenarios.

A major downside of the DSAP+ model is its computational complexity. The problem formulation at its core is NP-complete. In simulations, we could solve only very small problem instances with 20 to 30 VMs within one hour of computation using the Gurobi solver.

Chapter 5

Vector packing as static allocation controller

Many VMs exhibit resource utilization with a daily or weekly seasonality. Workload profiles describe an average season and help to refine the VM's demand characteristics compared to time-invariant reservation specifications like static CPU cores and memory. SSAPv [86] strongly increases consolidation density while maintaining service quality leveraging workload profiles.

In this work we show that heuristics are on par with SSAPv solutions if calculation time is constrained. Further, there are no significant differences between heuristics - simpler heuristics perform just as well as more complex ones. Also, solution quality does not increase with input sizes in excess of 200 VMs, allowing calculations to be clustered on a per rack basis. In scenarios with unlimited calculation time SSAPv still outperforms all heuristics.

First, we evaluate the effectiveness of well-known vector bin packing heuristics on daily workload profiles of enterprise data centers. To our knowledge, this is the first study to combine high-dimensional workload profiles with vector bin packing heuristics. In related work, vectors are used to represent single demand values for multiple resources. Second, we provide comprehensive guidelines for parametrizing vector bin packing heuristics for the purpose of

VM allocation. Third, we show that vector bin packing heuristics can substitute integer programs if solution time is constrained and workload profiles of similar dimensionality are used.

Texts in this chapter are based on a previous publication [95].

5.1 Problem formulation

We ask for an VM allocation which assigns workload profiles to servers without oversubscribing any of these. We consolidate servers based on a single restrictive resource, the CPU. To maximize a server's energy efficiency it is necessary to fully utilize each CPU as under-utilization leads to lower efficiency as shown in Section 1.3.

A workload profile describes a VM's average utilization of a 24 h day for a single resource in τ periods of equal lengths. A vector $\vec{d_j}$ with $\tau = dim(\vec{d_j})$ values thus describes the CPU utilization of a VM j over all τ periods, with one value per period.

A server i is represented by a vector of residual capacities $\vec{r_i}$ with dimensionality τ. It explains the amount of free server resources for each period. Whenever a VM j is assigned to a server i, its load vector is subtracted from the server's residual capacity vector $\vec{r_i} - \vec{d_j}$. If one vector component turns out to be negative, this particular server is oversubscribed. An allocation of VM j to server i is valid if $d_j^t \leq r_i^t$ for $1 \leq t \leq \tau$.

We search a valid allocation for a given set of VMs and their respective workload profiles with the objective of minimizing the number of servers, no oversubscriptions allowed. This problem is known as the multidimensional vector bin packing problem [29].

5.2 Existing approaches

Various works on integer programs to solve the multidimensional vector bin packing problem exist [2, 89]. Speitkamp et al. [86] recommend workload profiles with less than 24 dimensions, arguing that downsampling workload profiles hurts solution quality only marginally and reduces major complexity hurdles.

Setzer et al. propose a novel approach to cut complexity [77]. SVD is leveraged to extract workload characteristics used by an integer program that originated from the one proposed by Speitkamp. This approach combines the advantages of using workload profiles with a reduction in computational complexity.

Panigrahy et al. [90] propose a vector bin packing heuristic for server consolidation based on the dot product of two vectors. Li et al. [52] subsequently showed that the cosine of two vectors yields slightly better results. Both compared their approaches to existing heuristics such as First-Fit or the euclidean distance metric as proposed by Srikantaiah et al. [87]. In these and other studies [89], vector components represent singular demand values for server resources, for example CPU or memory.

The vectors in this study instead represent the workload profile of a single constrained resource (CPU) over time, where time is partitioned into τ equal-length periods. Compared to previous studies, vectors used by this approach are of higher dimensionality. To our knowledge, this is the first study combining vector bin packing heuristics with workload profiles to calculate dense allocations for a given set of VMs.

The approach closest to ours is by Maruyama [54], who propose a generic algorithm for vector bin packing. They evaluate the impact of different parameters by conducting simulations founded on synthetic data, where vector elements are randomly chosen from a bell shaped or skewed distribution. However, histograms drawn from representative workload profiles as shown in Appendix E suggest different distributions, leaving room to exploit complementarities and questioning the applicability of [54]'s approach.

5.3 Simulation setup

Our simulations make use of precalculated workload profiles to evaluate the effectiveness of vector bin packing heuristics for allocating VMs to servers. These workload profiles stem from two large European IT service providers and are not generated in a synthetically manner as done by other studies. An extensive descriptive statistical analysis is provided in Appendix E.

For each of the 451 utilization traces we calculate a profile closely following the approach described in Appendix E. We select the CPU as the single limiting resource, and ignore other resources, since the scalability of CPU power is inferior to the scalability of all other server resources. The majority of the time series exhibit a daily load pattern if weekends are excluded.

We created twenty sets of workload profiles. For each set, 180 workload profiles were randomly picked from the 451 workload profiles. Each simulation treatment was replicated twenty times, once for each set.

Simulations were conducted as full factorial experiments with three factors. The factor scale describes the number of VMs to allocate with levels 30, 90, 120, 150, and 180. The factor profile size τ describes the workload profile downsampling with levels 288 (no downsampling), 24 (one per hour), 12 (one per two hours) and 6 (one per three hours). We implemented controllers with four bin packing heuristics and one integer program as the third factor, algorithms, with five levels: 1) FF (First-Fit), 2) FFv (First-Fit with variable workload), 3) DotProduct [90], 4) Cosine [52], and 5) SSAPv [86]. We shortly summarize the working of each controller.

FF is a one-dimensional bin packing heuristic and therefore needs to extract a single scalar value from the VM's workload profile vector, in this case the vector component with the maximum value. It then places the VM on the first server that has enough residual capacity to accommodate the VM. First-fit with variable workload (FFv) on the other hand does a component-wise comparison of the workload profile and the server's vector of residual capacities and then similarly places the VM on the first server that can host the VM.

SSAPv [86] is an integer program to calculate an allocation with the objective of minimizing the number of servers, which we solved using the Gurobi solver. We limited its computation time to 25 min for small scenarios with up to 60 VMs and to 45 min for larger scenarios with up to 180 VMs. Gurobi was able to produce valid, yet potentially suboptimal solutions for almost all treatments. Since our study is about the practical applicability of the described approaches, we argue that data center operators in operational environments limit the calculation time of solvers like Gurobi. We therefore compare Gurobi's (sub)optimal solutions with vector bin packing heuristics.

Panigrahy et al. [90] propose the dot product of the the server's residual capacity and the VM's resource demand vectors as the metric for determining the allocation target. For each new VM the dot product of a server's weight $\vec{w_i}$, residual $\vec{r_i}$, and the VM's demand $\vec{d}$ is calculated as shown in Equation 5.1. The VM is placed on the server i with the greatest dot product.

$$\sum_{t=1}^{\tau} w_i^t r_i^t d^t, \forall i \in I \tag{5.1}$$

Xi [52] et al. instead propose the minimum cosine (see Equation 5.2).

$$\cos(\vec{r_i}, \vec{d}) = \frac{\vec{r_i} \cdot \vec{d}}{|\vec{r_i}||\vec{d}|}, \forall i \in I \tag{5.2}$$

5.4 Simulation results

We conducted simulations in a full factorial experimental design. Each treatment was replicated twenty times, once for each profile set. Simulations were conducted in fully randomized order.

The effectiveness of our vector bin packing controllers was measured by their allocation density $d = {}^{n}\!/_{m}$, where n denotes the number of VMs to allocate and m the server demand. Since solutions obtained with the Gurobi solver were time-constrained and thus potentially suboptimal, we were unable to use

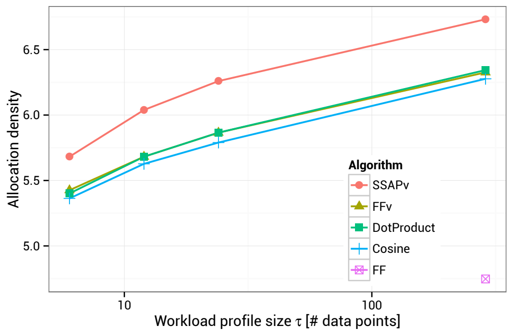


Figure 5.1: Allocation density vs. profile size at a scale of 120 VMs

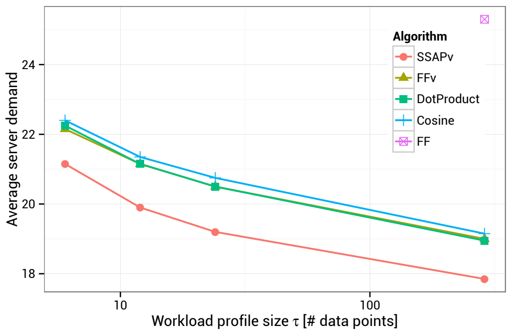


Figure 5.2: Required servers vs. profile size at a scale of 120 VMs

the competitive value (i.e. heuristic vs. optimum solution) as metric. We now describe the simulation results in more detail.

5.4.1 Effect of profile size

Profile sizes with levels 6, 12, 24, and 288 were used for each heuristic. The factor scale, i.e. the number of VMs, was fixed at level 120. Figure 5.1 and 5.2 depict algorithm performance with respect to density d and the number of required servers. FF is represented by a single point owing to its independence of profile size.

As expected, SSAPv delivers the highest allocation density and lowest server demand regardless of profile size. FF on the other hand yields the worst results regardless of profile size. Evidently, vector bin packing heuristics successfully exploit profile complementarities without much difference between them. DotProduct and FFv slightly outperform Cosine, which does not necessarily contradict the results of Xi et al. [52] as our vectors are of higher dimensionality.

Allocation density improves with increasing profile size: going from 6 to 24 yields the greatest improvement. This is consistent with the results of Speitkamp et al. [86] with regard to SSAPv. The findings disagree with those

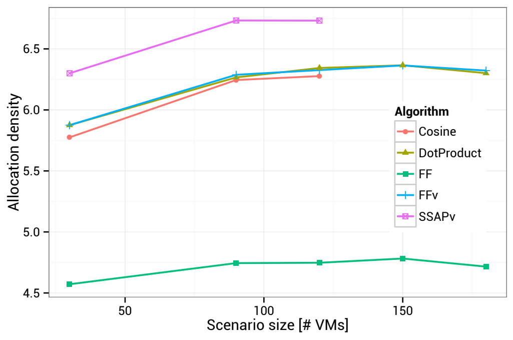


Figure 5.3: Allocation density in relation to scenario scale at a profile size of 288 values

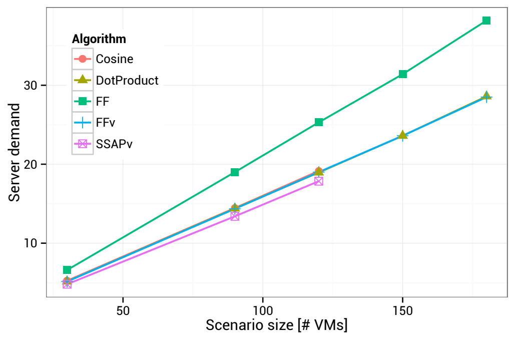


Figure 5.4: Required servers in relation to scenario scale at a profile size of 288 values

of Talwar et al. [90], as increasing profile size and thus vector dimensionality have a positive effect on density.

SSAPv delivers the best allocation density but takes much longer to calculate solutions. For treatments with a profile size of 288 and 120 VMs we limited calculation time to 45 min. In contrast, no heuristic took longer than 5 s to calculate the allocation without optimizations done to the Python code base.

5.4.2 Effect of scale

In this set of simulations we wanted to analyze the effect of scenario scale on allocation density and server demand. We fixed the profile size to 288 values. The scale was varied between 30, 90, 120, 150, and 180 VMs. There is missing data for SSAPv as Gurobi was unable to provide valid solutions within 45 min for treatments in excess of 120 VMs.

Results are depicted in Figure 5.3 and Figure 5.4. SSAPv again outperforms all heuristics with regard to server demand and allocation density, while the one-dimensional FF heuristic produces the worst overall results.

The required number of servers increases almost linearly with the number of VMs for all heuristics and SSAPv. This is different from an increase in profile size, where server count decreases in a non-linear way. The slope of vector bin

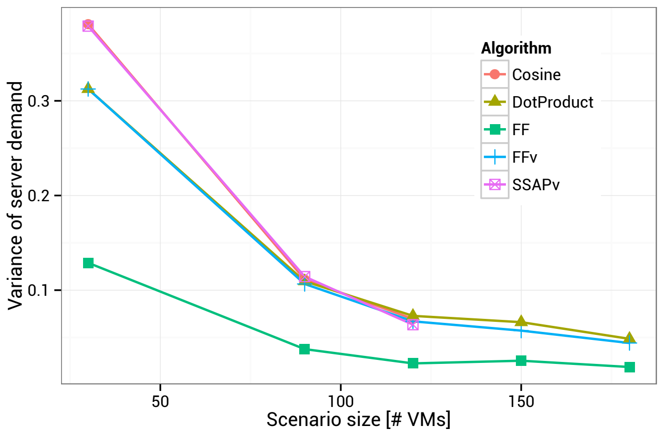


Figure 5.5: Variance of the number of required servers over 20 experimental replicas in relation to scenario scale at a profile size of 288 values

packing heuristics is about 0.15 and slightly greater than the one of SSAPv with 0.145. FF has the greatest slope with 0.21 which is especially bad for very large scenarios.

Increasing scale from 30 to 120 VMs again positively affects allocation density for all approaches as shown in Figure 5.3. This makes sense as larger scenarios should provide more complementarities between profiles. Increasing scale beyond 120 VMs does not increase allocation density further. We conclude that there is no benefit in simulations for larger scenarios.

We examined the variance of the number of required servers and scale between simulation replicas. We found allocation density to be low if only a few VMs are used. As shown in Figure 5.5 the variance strongly decreases with scale going from 30 to 120 VMs and remains approximately constant for larger scenarios. This is likely because small scenarios do not provide enough complementarities that can be leveraged by vector bin packing algorithms or SSAPv.

This result suggests scenarios with a scale between 90 and 200 VMs. Smaller scenarios produce less dense allocations with a high variance of the number of required servers. Depending on the (random) choice of VM types and thus the ability to exploit complementarities, allocations can benefit or hurt accordingly.

Increasing scale above 200 VMs does not improve allocation density much, but takes longer to solve, especially for SSAPv. Although SSAPv provides better results, its practical applicability is questionable as solvers consume large amounts of time and might not even return valid results within bounded time.

5.4.3 Interaction of scale and profile size

Our findings show that increases in scale or profile size lead to denser allocations and thus decrease the number of required servers. We now pose the question of interaction effects between these two factors. Current results suggest that a simultaneous increase in both factors yields the densest allocations. However, adverse interaction effects might lead to less dense allocations.

We thus conducted an ANOVA analysis based on our simulation results. We set density as the target variable and included the two factors profile size (levels: 6, 12, and 24) and scenario scale (levels: 30, 90, and 120), resulting in a full factorial 3^2 design. Refer to Table 5.1 for the results of the two-way ANOVA analysis.

As expected from our chart analysis, both factors are significant at $p \leq 0.001$. Interaction between profile size and scenario scale is not significant with $p = 0.773$. In other words, increasing both factors yields the best allocation results.

The plots of residuals versus fitted values and residuals versus run order of treatments did not exhibit any heteroscedasticity, clear pattern, or trend. We found a single outlier in the residuals. Excluding this outlier had no effect on the ANOVA results. The normal probability plot reveals a deviation from the normality assumption, indicating a slightly skewed distribution that should not have any impact on the results.

5.4.4 Comparing algorithms

As the previous charts show, vector bin packing heuristics produce very similar results, with FF consistently performing worse and SSAPv better. To further

	SumSq	MeanSq	Fvalue	Pr(>F)
VM Count	15.36	15.361	105.157	<2e-16***
Profile Size τ	27.56	13.780	94.329	<2e-16***
VM Count:Profile Size τ	0.08	0.038	0.257	0.773
Residuals	104.30	0.146		

Table 5.1: ANOVA results for factors scenario scale and workload profile size in relation to allocation density

	diff	lwr	upr	padj
DotProduct-Cosine	0.06666667	-0.1472012	0.2805345	0.9082794
FF-Cosine	-1.52933552	-1.7432034	-1.3154676	0.0000000
FFv-Cosine	0.04912281	-0.1647451	0.2629907	0.9683139
SSAPv-Cosine	0.45473979	0.2408719	0.6686077	0.0000005
FF-DotProduct	-1.59600218	-1.8098701	-1.3821343	0.0000000
FFv-DotProduct	-0.01754386	-0.2314117	0.1963240	0.9993892
SSAPv-DotProduct	0.38807312	0.1742053	0.6019410	0.0000211
FFv-FF	1.57845833	1.3645905	1.7923262	0.0000000
SSAPv-FF	1.98407531	1.7702074	2.1979432	0.0000000
SSAPv-FFv	0.40561698	0.1917491	0.6194849	0.0000082

Table 5.2: TukeyHSD comparing allocation controllers on allocation density

verify this finding we conducted a one-way ANOVA analysis on the allocation density. The factor algorithm had five levels: FF, FFv, DotProduct, Cosine, and SSAPv. The data for a scenario of scale 120 VMs and a profile size of 288 values was used for analysis. The results of the TukeyHSD test are shown in Table 5.2.

The residual plots of residuals versus fitted values and residuals versus run order of treatments did not show any heteroscedasticity, clear pattern, or trend. Normal probability plot was unobtrusive again.

SSAPv is significant compared to any of the heuristics with p-values < 0.001. FF is significant compared to any of the algorithms as well. The results are consistent with our previous findings gained by chart analysis.

There is no significant difference between vector bin packing heuristics. The more advanced heuristics DotProduct and Cosine have no significant advantage over the straightforward FFv heuristic. Again, this result is in line with our expectations from chart analysis. For low dimensional vectors [90] and [52] showed that DotProduct and Cosine outperform FFv. At least for high

	diff	lwr	upr	padj
DotProduct-Cosine	0.06666667	-0.1464126	0.2797460	0.9071496
FF-Cosine	-1.52933552	-1.7424148	-1.3162562	0.0000000
FFv-Cosine	0.04912281	-0.1639565	0.2622021	0.9678878
SSAPv-Cosine	-0.01754386	-0.2306232	0.1955355	0.9993802
FF-DotProduct	-1.59600218	-1.8090815	-1.3829229	0.0000000
FFv-DotProduct	-0.01754386	-0.2306232	0.1955355	0.9993802
SSAPv-DotProduct	-0.08421053	-0.2972898	0.1288688	0.845066298
FFv-FF	1.57845833	1.3653790	1.7915376	0.0000000
SSAPv-FF	1.51179166	1.2987123	1.7248710	0.0000000
SSAPv-FFv	-0.06666667	-0.2797460	0.1464126	0.9071496

Table 5.3: TukeyHSD comparing SSAPv (profile size 24) with heuristics (profile size 288) with regard to allocation density

dimensional vectors, we could not confirm this finding.

Despite significant differences between SSAPv and vector bin packing heuristics, these might be irrelevant in practical applications. The results shown in Figure 5.4 show that SSAPv and vector bin packing heuristics produce almost identical results if values are rounded to the next integer. We should stress once more that a time limit was imposed on the Gurobi solver. Given unbounded time, SSAPv solutions may still improve.

To contain solvers with respect to calculation time for large scenarios, the SSAPv is often parametrized with shorter profile sizes as proposed by [86]. We pose the question of how well heuristics with profile sizes of 288 perform compared to SSAPv solutions based on a downsampled profile size of 24.

We conducted an ANOVA analysis to compare the allocation density of SSAPv with heuristic solutions, with profile sizes of 288 for the heuristics and 24 for the SSAPv. Table 5.3 shows the results of the TukeyHSD test.

In this scenario, SSAPv does not perform significantly better than vector bin packing heuristics. Similar to our previous ANOVA analysis, FF performs significantly worse than any other approach including SSAPv.

The finding is practically relevant, since it indicates that heuristics perform just as well as the time-constrained SSAPv solver, especially if the time limit is in the order of seconds. While heuristics can be parametrized with high profile sizes, SSAPv has to be solved for small profile sizes to curb calculation time.

The trade-off obviously decreases SSAPv solution quality to a point where it performs just like heuristic solutions.

We found that the 99th percentile of execution time for a scale of 120 VMs and a profile size of 288 values amounts to 3.6 s for heuristics and 30 min for the SSAPv. Most of the SSAPv solutions were not proven optimal by Gurobi.

5.5 Service quality

Penalties usually take effect if service quality goals are not fulfilled. Therefore, the quality of service provisioning is an important aspect for data center operators. Often, VMs are oversized to handle unexpected load spikes. Profile based consolidation as proposed by this work pursues the goal of eliminating entirely such security buffers.

Still, security buffers can go side by side with profiles. Instead of overprovisioning resources, security buffers can be incorporated into profile calculation. It can be calculated such that 100 % of past loads are coverd. An additional security buffer can be added to profile values that exhibit a high demand so that buffer size is varied with expected demand or expected demand variance. Alternatively, a profile could include only 90 % of the past load demands. This would be a more aggressive profile which would result in more aggressive allocations.

One might expect that slight deviations of the load behavior in an aggressively calculated profile result in a decreased service quality. However, this is not the case in realistic environments. In Chapter 2 we analyzed the performance of consolidations calculated by the SSAPv in a testbed infrastructure. It showed that even significant differences between the actual load and an aggressively calculated workload profile only causes a slight decrease in service quality. Taking into account that vector bin packing heuristics on average tend to calculate conservative allocations mitigates the concern even more.

5.6 Conclusions

We analyzed one bin packing and three vector bin packing heuristics alongside an SSAPv integer program to allocate VMs to a minimum number of servers based on VM workload profiles. Our analysis leads to a number of guidelines that may benefit data center operators as well as software engineers facing the problem of implementing static controllers.

First, allocations for small scale infrastructures with less than 50 VMs should be calculated using SSAPv or other integer programming techniques. They deliver significantly better results than all heuristics we have tested. Modern integer program solvers frequently find a viable solution within a couple of minutes.

Second, vector bin packing outperforms one-dimensional packing but there are no significant differences between vector bin packing heuristics. We compared four packing heuristics ranging from the very simple, one-dimensional FF to ones that leverage vector algebra. FF reduces the workload profile of a VM to a single scalar value, performing significantly worse than FFv or any other vector bin packing heuristic leveraging workload profiles with profile sizes between 6 and 288.

Third, including more than one server rack consisting of approximately 40 servers and 200 VMs into a single calculation does not improve allocation density. While SSAPv and heuristic solutions improved with scale increasing up to 180 VMs, increasing scale even further did not yield significantly better results (or worse, for that matter). Instead, calculation time increases significantly for the SSAPv. We therefore recommend per-rack calculations.

Fourth, vector bin packing heuristics should be parametrized with detailed workload profiles of many values. If the profile size is increased from 6 to 288 values, allocation density increases as well. The biggest improvement is noticeable between 6 and 24 values. Beyond that, allocation density increases only slowly.

Fifth, medium and large scale scenarios should be calculated using a heuristic if calculation time is constrained. While such scenarios can still be solved by integer program solvers, the problem complexity and in turn the profile size have to be reduced to obtain results within minutes. However, going below profile sizes of 24 negatively affects allocation density. We found that heuristics running with highly detailed profiles on large scenarios produce the same allocation quality as SSAPv running on downsampled, low-dimensional profiles.

Chapter 6

Colocating VMs on main-memory database servers

Previous chapters aimed at increasing the average server utilization. This was achieved by allocating VMs to servers so that a minimum number of servers was required. Servers were provisioned with bare-metal hypervisors that run VMs exclusively. Database servers are typically the core-components in enterprise IT systems and today many databases are already deployed in VMs. Running emerging main-memory database systems within VMs causes huge overhead, because these systems are highly optimized to run on a bare metal hardware.

Prof. Thomas Neumann demonstrated[1] that virtualization may reduce TPC-C throughput by 33 %. However, running these systems on bare metal servers results in low resource utilization, because database servers are often sized for peak loads, much higher than the average load. Instead, we propose to deploy them within light-weight containers. Linux Containers (LXCs) provide a light-weight form of process encapsulation and allow to limit resource usage of a process group. On the one hand, resource limits of containers can be changed quickly. On the other hand, containers do not incorporate significant

[1]The talk with the title "Scalability OR Virtualization" took place on November 18th 2011 at "Herbsttreffen der GI-Fachgruppe Datenbanksysteme" in Potsdam, Germany. The talk was in German, but the slides are in English. A video recording is available online `http://www.tele-task.de/archive/series/overview/874/`.

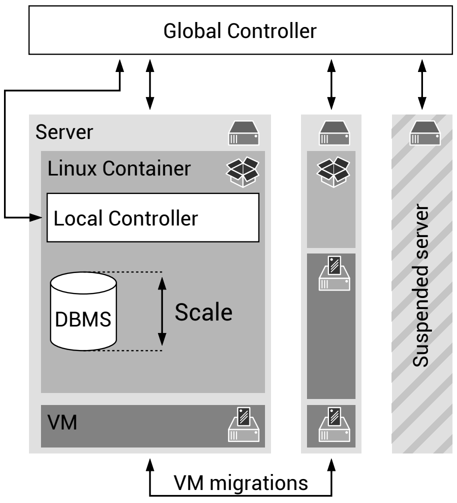


Figure 6.1: Cooperative resource allocation control approach

overhead, as all processes are still managed by the Linux Kernel of the host server. As of now it is impossible to migrate containers in the same way as VMs. However, main-memory database management system (DBMS) are poor candidates for migration anyway, due to high memory demand and data access frequency. We propose to improve resource utilization by temporarily running other applications on the database server encapsulated by VMs. Servers on which these VMs would normally run can be suspended to decrease average server demand.

Figure 6.1 shows a server farm consisting of several servers that run VMs of various sizes. The size of a VM represents its current resource demand. As discussed above, we do not run the DBMS in a VM, but in a container. The assignment of VMs to servers is determined by a central control unit called GlobalController. It may assign several VMs to the same server and change the assignment using VM migration.

In many cases, VMs do not fully utilize their resource reservations. Static or dynamic controllers as discussed in Chapters 2 through 4 can be used to reduce the server demand by increasing average server utilization. These approaches work for database systems too, as shown by Graubner et al. [36]. It is undesirable to operate servers at a medium utilization level, as the energy demand depends linearly on the server's load with a high intercept as discussed in Section 1.3. The two most power efficient operating modes are suspend and full utilization [6].

Accurate estimates on resource demand are required to maintain SLAs while resource utilization is improved. SLA-based resource allocation [35] is a topic of active research in the cloud computing community. Furthermore, current database systems do not handle dynamic changes of their assigned resource capacity well. We have observed that resource assignments should not be reduced without taking precautions and that dynamically added resources are often not used right away. If resource allocation is changed dynamically without reconfiguring the DBMS accordingly, DBMS performance may deteriorate disproportionately. Furthermore, long-running queries may not be able to use the additional resources during runtime as query operators were instantiated with a fixed number of processing threads and memory at the time when they were started. To our knowledge there are no commercial DBMSs that support dynamic reconfiguration of critical resources like memory or CPU cores within seconds while the database is running. Therefore, we focus on emerging main-memory DBMSs, that are more flexible with regard to resource allocation changes and that support the mixed workloads of today's operational (real-time) business intelligence applications.

We propose an cooperative approach in which the DBMS communicates its resource demand, gets informed about currently assigned resources and adapts its resource usage accordingly. First, the GlobalController may change the actual resource allocation of an application at runtime by resizing its container, as illustrated in Figure 6.1. Second, communication allows applications to resize their resource usage according to the actual resource allocation of their VM. For example, a database system may execute queries on-disk instead of in-memory in presence of memory shortage. Third, applications may be able to estimate their resource requirements quite accurately and provide this information to the GlobalController in addition to existing workload traces. This increases the data quality available to the GlobalController as e.g. application buffers and queues are included in this data. Ultimately, cooperation may help to improve consolidation efficiency.

An alternative to virtualization would be to consolidate DBMSs with complementary resource requirements by employing multi-tenancy techniques as discussed by Bobrowski [13]. Moreover, advanced multi-tenancy techniques can be employed to further reduce space and performance overhead of consolidation as shown by Aulbach et al.[4]. Soror et al. [84] have shown that databases with complementary resource requirements can be consolidated using virtualization. However, our proposed approach allows a combination with arbitrary applications running in VMs, which is a very attractive scenario especially for IaaS cloud providers.

For evaluating cooperative control, we built an experimental testbed that is based on latest virtualization technology (Linux kernel-based virtual machine (KVM) and LXC) and an emerging main-memory DBMS called HyPer [45], that has been developed at the chair of Prof. Kemper and that achieves outstanding online transaction processing (OLTP) and online analytical processing (OLAP) performance numbers. We extended HyPer with a LocalController component, that enables the DBMS to adapt to externally given resource assignments and delivers resource demands to the GlobalController. Based on this setup we conducted experiments to evaluate the effects of cooperative control, analyzed the performance impact when spare resources of an HyPer database server are used to host virtual machines and monitored SLAs compliance.

Texts in this chapter are based on a previous publication [74].

6.1 Dynamic resource allocation of main-memory DBMSs

Main-memory DBMS like HyPer and SAP Hana [27] allow to process business logic inside the DBMS. Thereby performance may be improved significantly, as the number of round-trips between application servers and the DBMS can be reduced. HyPer minimizes synchronization overhead by processing transactions sequentially according to the one-thread-per-core model. As transactions are processed sequentially, incoming transactions have to wait until preceding transactions have been processed. For high and bursty arrival rates, execution times of individual transactions thus have to be very low in order to achieve low response times. This is achieved by keeping all data in memory and minimizing synchronization overhead by processing transactions sequentially. With this approach low response times and extremely high throughput rates can be achieved. But we need to ensure that the memory allocation is never reduced below a certain lower bound so that the whole dataset fits into memory and swapping is avoided. This requirement needs to be considered when allocating VMs on a main-memory database server. The lower bound depends on two factors: amount of memory needed for the actual data and the memory demand for processing business transactions. The former is typically large, but can be reduced by employing compression techniques. According to Hasso Plattner [66], compression rates of factor 20 can be achieved for typical customer data using column-store technology. The latter is typically small, as only one transaction per core is processed concurrently in the one-thread-per-core model. If memory allocation would be reduced below this lower bound during runtime, severe interferences are highly probable. The operating system will start to swap data from memory to disk and the execution time of a single transaction would increase dramatically if it touches data that is not available in memory. Thus, pure OLTP workloads do not leave much room for improvement.

But HyPer supports mixed workloads consisting of interaction-free[2] transactions (OLTP) and read-only analytical queries (OLAP) on the same data and thereby enables business applications with operational (or real-time) business intelligence features. For OLTP, we need to ensure that memory allocation is not reduced below a certain lower bound, as discussed above. For mixed workloads, this lower bound is a bit higher, because additional memory is required for snapshots of the data, as HyPer processes read-only analytical queries

[2]HyPer assumes that there is no user interaction during processing of single transactions, which is common for high throughput OLTP engines

in parallel on snapshots of the data. But HyPer uses special techniques for minimizing the memory and processing overhead of these snapshots[3]. Apart from the snapshots, analytical queries require memory for intermediate results, which may be substantially large, e.g. queries involving large join operations or other pipeline-breakers that require to materialize large intermediate results. If arrival rates are high, it may be necessary to keep these intermediate results in memory in order to achieve required throughput rates. If query arrival rates are low, intermediate results could also be stored on disk, if expected response times are large enough to allow for the required disk I/O. For varying OLAP loads, it depends on the actual load if a lot of memory is required for processing lots of queries in memory or if there are only few queries that can be processed on disk. If the GlobalController would know about the current OLAP load situation and how it will change in the near future, memory allocation could be changed accordingly. But if the GlobalController changes memory allocation without this knowledge, analytical queries may miss their SLOs.

We analyzed how much execution time deteriorates due to OS-swapping, when the memory assignment is reduced while a query is executed. We used HyPer to analyze the execution time for joining order and order-line tables from the CH-benCHmark[4] with 500 warehouses. HyPer allows to specify how much memory may be used for a join query and we used a container to enforce further limits on resource usage.

Initially, we assigned sufficient memory to keep intermediate results completely in memory. When the memory assignment was reduced by 10 % shortly after query execution started, query execution did not terminate even after we waited several hours and we decided to abort the query. But even if we took the normal execution time as a baseline and reduced the memory assignment by 10 % only 1 s before the end of baseline execution time, the execution time

[3]The size of the snapshots depends on memory access patterns, as discussed in [73], and can be minimized by clustering the current data, that is still modified [57]

[4]CH-benCHmark [21, 30] analyzes the suitability of DBMSs for mixed workloads of operational (or real-time) business intelligence applications and combines transactional load based on TPC-C with decision support load based on a TPC-H-like query suite run in parallel on the same tables.

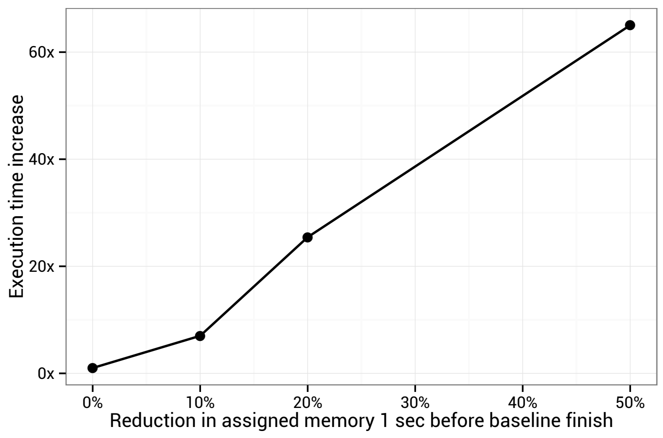


Figure 6.2: Reducing memory assignment to the database container during query execution

still increased by a factor of 7. Figure 6.2 shows that execution time grows when the memory assignment is further reduced, again 1 s before the end of the baseline execution time.

At a memory reduction of 50 % the execution time increased by factor of 65. The reductions in execution time also show the effectiveness of containers. The conclusions we draw from these experiments is that knowledge about how much memory is available to the DBMS is very important to choose the physical operators and their parameters right in order to utilize all available memory and to avoid OS-swapping. Furthermore, it is critical to maintain the lower bound of main memory, else transactions may have much longer execution times and probably will miss their response time goals, as transactions are processed sequentially. If the arrival rate is high, transaction requests may even be dropped, once the system runs out of resources for processing the request queue.

Apart from memory, processing resources are also very important for query processing in main-memory DBMSs. With modern virtualization techniques, the CPU shares and the number of virtual cores of a virtual machine can be changed at runtime. Providing an application with more memory can only result in improved performance if the larger data volumes in memory can be processed efficiently with the available CPU cores and CPU shares. Today, multi-core CPU architectures are common and DBMSs need to process queries in parallel in order to utilize these architectures efficiently.

Kim et al. [47] and Blanas et al. [11] investigate the parallelization of the probably most important DBMS operator, the join of two relations. They investigate different ways of adapting sort-based as well as hash-based join algorithms to modern multi-core environments and show that high performance gains can be achieved. However, join operators do not only have to exploit all resources available at the time they were instantiated, but they should be able to adaptively adjust the numbers of processing threads to changing resource allocations, including number of virtual CPU cores. Otherwise additional resources would be idle until potentially long-running queries are processed completely or new queries arrive.

If the number of CPU cores is reduced, but the join operator does not reduce its number of processing threads, therefore performance decreases, as the number of context switches strongly increases.

6.2 Cooperative consolidation

Emerging main-memory DBMSs, like HyPer, are designed for multi-core servers with huge amounts of memory. But resource utilization is typically low, as such database servers are often provisioned for peak loads and average load is much lower. Virtualization and server consolidation can be employed to improve resource utilization and reduce operational costs. However control mechanisms at the virtualization level and optimization mechanisms at the database level may interfere with each other, because local adaptive control within VMs and containers makes it difficult to monitor the load accurately and current DBMSs do not handle dynamic changes to resource allocation well.

In order to improve resource utilization for emerging main-memory DBMS, like HyPer, we propose a cooperative approach that enables server consolidation for emerging main-memory database systems. Communication between a

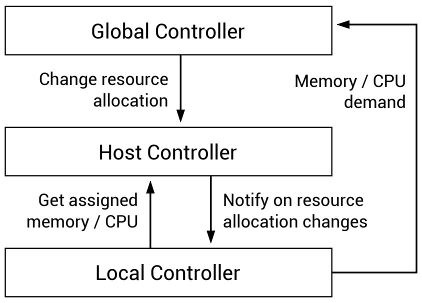

Global Controller
Change resource allocation
Memory / CPU demand
Host Controller
Get assigned memory / CPU
Notify on resource allocation changes
Local Controller

Figure 6.3: Communication between GlobalController and LocalController via a HostController

GlobalController and LocalControllers (see Figure 6.3) allows to avoid interferences between control mechanisms at the virtualization level and optimization mechanisms at the database level. DBMSs need to be aware of dynamic changes to resource allocation and coordination is needed in order to improve resource utilization while meeting SLOs.

6.2.1 Local and Host controller

As shown in Figure 6.1, the physical server resources are divided between all VMs and containers. We propose a cooperative approach where each VM runs a LocalController that communicates with a GlobalController, as illustrated in Figure 6.3. The GlobalController implements a dynamic allocation controller that is responsible for allocating VMs. It adapts to changing resource demands by migrating VMs under the objective of using as few servers as possible.

GlobalController and LocalController communicate via a HostController, that provides information on how much physical resources are currently assigned to a given VM and notifies the LocalController running inside a VM when its resource allocation changes. The LocalController for the DBMS controls the number of concurrently executed queries in order not to exceed the current resource assignment of the DBMS container but still meet the query SLOs.

The simplest possible implementation of such a query scheduling component is to use a first in first out (FIFO) queue. However, a FIFO queue may be suboptimal under the given circumstances, because FIFO executes queries in arrival order without considering that spare resources may be used by other VMs or containers running alongside. Thus, a FIFO scheduler could cause OS-swapping because it does not take the available memory resources into account, which negatively affects the query execution performance as described in Section 6.1.

A more advanced implementation *FIFO+* communicates resource requirements to the GlobalController. It takes the memory demand of OLAP queries as well as the available resources into account. The former, can be estimated quite accurately for main-memory DBMSs. The latter information is available from the GlobalController via the HostController. Based on this information, OS-swapping can be prevented, as queries are only processed in-memory if there are enough resources available.

Once resource allocation changes, the LocalController gets notified. When resource allocation is reduced, we propose to abort longer-running queries and to restart them with different parameters in order to avoid OS-swapping. There is related work in the area of workload management on how to manage long-running queries [49]. In contrast, we focus on how to prevent queries from taking unexpectedly long due to dynamic changes to resource allocation by restarting them right away when such changes occur. We focus on join operators, because joins are common database operations that often impact the runtime of query execution plans significantly and potentially require large intermediate results.

For join queries, there is a big difference between in-memory and on-disk execution times. In order to demonstrate this, we use the same join from CH-benCHmark as in Section 6.1 and vary the amount of available memory. Figure 6.4 shows that there is a huge spike in execution time if intermediate results do not fit completely in memory. Just after the 100 % mark on the x-axis denoting the quotient of required memory and assigned memory, only little extra memory would be required to keep intermediate results completely in memory, but still execution time increases by a factor of 17 — from 19 s to 5 min [5].

[5]In order to analyze possible overheads caused by LXC, we repeated the measurements

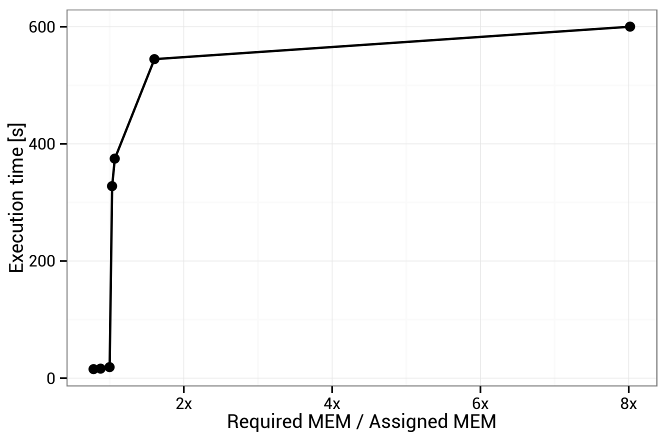


Figure 6.4: Execution times of in-memory vs. on-disk queries

HyPer used the hash join algorithm in this experiment, which is a common join algorithm for main-memory DBMS. The results suggest that join queries should be processed either completely in-memory or on-disk with only little memory, as more memory does not improve execution time much as long as the join cannot be processed completely in memory. There are other join algorithms, like hybrid hash join [24], that use additional memory more effectively and could level off the spike a bit.

Due to the large difference in execution times, it may make sense to abort longer-running queries that involve joins and to restart them with different parameters in order to reduce response times and improve throughput, once resource allocation is increased. For example, if a join query, that was started on-disk using an external join algorithm, could now be processed in-memory, overall execution time (including aborted partial execution) may be lower than on-disk execution time. The LocalController has to decide according to a policy whether to wait until running (join) queries finish or to abort and restart

without LXC. When sufficient memory was assigned to keep intermediate results completely in memory, response times were basically the same as with LXC. For lower memory assignments, response time without LXC was better, but this was expected, as HyPer uses memory-mapped files to store intermediate results on disk. Without LXC, these files are buffered in memory and may even be prefetched by the Linux kernel. We have to prevent this behavior, as we want to use these spare resources for other purposes. Thus, increased response times for on-disk execution only show the effectiveness of LXC.

them. A simple LocalController policy is shown in Algorithm 4. If the memory assignment has been increased significantly or sufficiently such that running (join) queries can now be executed completely in-memory, the running (join) queries are aborted and restarted. If the memory assignment has been decreased such that more memory would be used than is available, running (join) queries are aborted in order to avoid OS swapping.

```
if $\mathsf{assignedMem} > \mathsf{lastMemAssignment}$ then
    if $\neg\mathsf{inMem}$ then
        if $\mathsf{assignedMem} \geq \mathsf{requiredForInMem}$ then
            StopAndRestart()
        else if $\frac{\mathsf{assignedMem}-\mathsf{lastMemAssignment}}{\mathsf{requiredForInMem}} > 10\,\%$ then
            StopAndRestart()
else
    if $\mathsf{assignedMem} < \mathsf{usedMem}$ then
        StopAndRestart()
end
```

Algorithm 4: LocalController policy

To demonstrate the potential of this Stop+Restart approach, we extended HyPer with a LocalController that implements this policy, admits at most one OLAP query at a time and requests the required resources for processing the query completely in-memory right before processing the query only with currently available resources. We combined this LocalController with a GlobalController that checks for resource requests once a second and assigns requested resources right away. Again, containers were used to change resources assignments quickly with low overhead. The example workload consists of seven join queries with varying resource requirements. We use the same join query as before, but with varying number of warehouses (500, 250, 500, 500, 250, 750, and 750). HyPer used the hash join algorithm in this experiment, which is a common join algorithm for main-memory DBMS and allows to estimate memory requirements for in-memory execution quite accurately, as the intermediate result consists mainly of a hash table on the smaller join input.

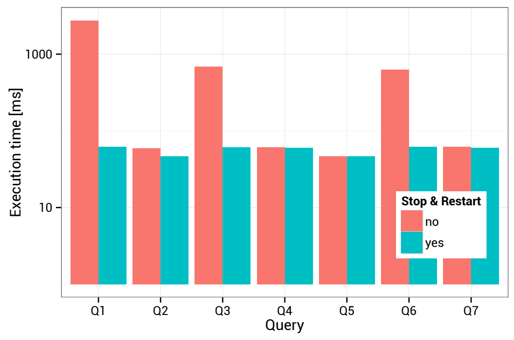


Figure 6.5: Execution times of different queries with and without Stop+Restart LocalController policy

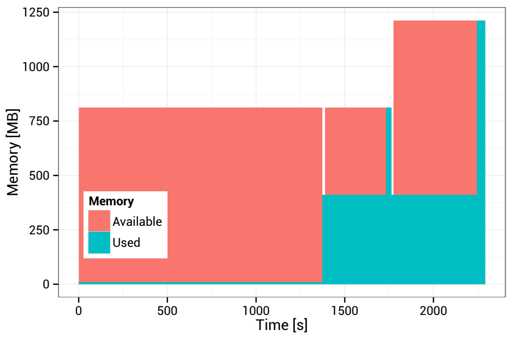


Figure 6.6: Memory utilization without Stop+Restart query execution

Figure 6.5 shows a comparison of execution times (normalized by number of warehouses) with and without the Stop+Restart approach. The initial resource allocation is one CPU core and 10 MB of memory. For the first join query overall execution times (including aborted partial execution) differ by a factor of 44, because without Stop+Restart it is executed on-disk (with very little memory) instead of completely in-memory. But also for the third and the sixth join query, there is an order of magnitude difference in execution times, because the preceding join query had lower memory requirements and therefore the GlobalController reduced the assignment accordingly. Without Stop+Restart, join query three and six were executed with only half and one third of the memory required for in-memory execution.

Figure 6.6 shows the utilization of the assigned memory without Stop+Restart. There are long delays until assigned resources are actually used. Although, additional resources are available quickly, as the GlobalController assigns requested resources right after processing the resource request. But these resources are not used right away without Stop+Restart. Coordination can help to improve utilization of dynamically assigned resources. If spare resources are actually used for other purposes, there may be delays for fulfilling increased resource requests, as discussed below and analyzed in the experimental evaluation.

6.2.2 Global controller

The GlobalController can use spare resources on database servers by migrating VMs towards them. For our experimental setup we implemented a GlobalController whose details are discussed in the following. The implemented GlobalController offers a network interface which receives stateless messages. A message consists of two fields: Memory and CPU. The memory-field describes the amount of requested memory in megabyte and the CPU-field describes the requested number of CPU cores. The GlobalController continually checks whether the memory and CPU assignment of the database system needs to be changed and whether it is possible to reuse spare memory for the allocation of VMs. It is important to notice that the GlobalController always assigns all the spare memory to the database system, even if the current memory requirements of the database are lower[6].

On the one hand, the database memory demand may increase. If the database server still has enough spare memory, the request is immediately fulfilled by the GlobalController. If there is not enough spare memory, the GlobalController needs to migrate one or multiple VMs from the database server towards another server, in order to increase the spare memory until the resource request of the database can be fulfilled. The database is notified by an update message once enough spare resources are made available and assigned to the database.

The memory demand of the database may decrease as well. In this case, the GlobalController increases the amount of spare memory immediately and checks whether it is possible to reuse spare memory to allocate VMs. If this is possible, it sequentially migrates VMs from other servers towards the database server until the spare memory is exhausted.

The GlobalController is responsible for the resource assignment and therefore has to select VMs to migrate towards and from the database server. There are

[6]The reason is that we execute migrations in a sequential manner. Therefore, the GlobalController is blocked during migrations and cannot handle incoming resource requests. By always assigning all spare resources to the database system we ensure that this is no issue. Even if the GlobalController would handle requests during a migration it could not assign more resources then it already has assigned.

multiple ways to decide which VM to pick for a migration. We implemented the First-Fit heuristic which is easy to explain. It always choses the VM that fits the requirements first. When there is spare memory, the controller picks the first VM which fits the spare memory as long as there is any. If a VM needs to be offloaded in order to make more memory available, the controller choses the first VM which would establish the desired state.

6.3 Experimental evaluation

Experiments were conducted on a server infrastructure that closely resembles the technological setting of IaaS cloud providers. Two budget servers, equipped with 16 GB of RAM, one Intel Xeon E5520 CPU with disabled hyper-threading and one 1 TB SATAII 7200 RPM disk was used to host VMs. Furthermore, VM images resided on a NFS volume of a separate storage server connected via a 1 Gbit Ethernet connection. In all experiments, there where 15 VMs in the system with four virtual CPU cores each and the following memory reservations: five VMs with 256 MB, five VMs with 512 MB, three VMs with 1024 MB and two VMs with 2048 MB. The amount of CPU cycles allocated to the VMs is weighted based on the size of the VM ($^1/_{64}$ per 256 MB). In our scenario, VMs can run arbitrary applications and we use the *stress*[7] utility to generate load on each VM. This utility simulates a hot memory working set by allocating memory (e.g. 20 MB) and changing parts of the memory as fast as possible. By that, migration times of VMs are significantly increased, as described by Clark et al. [60].

At the beginning of each experiment, 8 GB are reserved on the first server — the database server — to hold the database in a compressed format and all VMs are running on the second server — the swap server. First, no cooperative control is used, so that all VMs remained on the swap server and the database has all the resources available all the time. Second, cooperative control is activated and the GlobalController migrates VMs between the two servers in

[7]`http://weather.ou.edu/~apw/projects/stress/`

order to use spare memory on the database server. We analyze how much of the spare memory could be recycled by VMs and how the SLA violations and response times of the database queries change due to that.

No workload traces of business applications with operational (or real-time) business intelligence features are available so far, that capture the characteristics of emerging main-memory DBMS with efficient support for mixed workloads (OLTP and OLAP on the same data). However, database servers are always sized to handle peak loads without SLAs violations. Based on this, two very different workload scenarios were used.

In the first workload, queries arrive in five bursts (B1, B2, B3, B4, and B5) that occur in 5 min intervals and are executed in an endless loop. There are three query types (Q1, Q2, and Q3), corresponding to the join query described in Section 6.1, but with different number of warehouses (250, 500, and 750). The query mix of the bursts is as follows: $B1 = (4 \cdot Q1 + 12 \cdot Q2 + 44 \cdot Q3), B2 = (2 \cdot Q1 + 6 \cdot Q2 + 22 \cdot Q3), B3 = (1 \cdot Q1 + 3 \cdot Q2 + 10 \cdot Q3), B4 = (1 \cdot Q3), B5 = (80 \cdot Q3)$. This workload scenario was picked, because the bursts lead to a peak in the database memory demand. Such a memory-peak cannot be fulfilled by the GlobalController instantaneously as it has to migrate VMs away first, which is the worst case scenario.

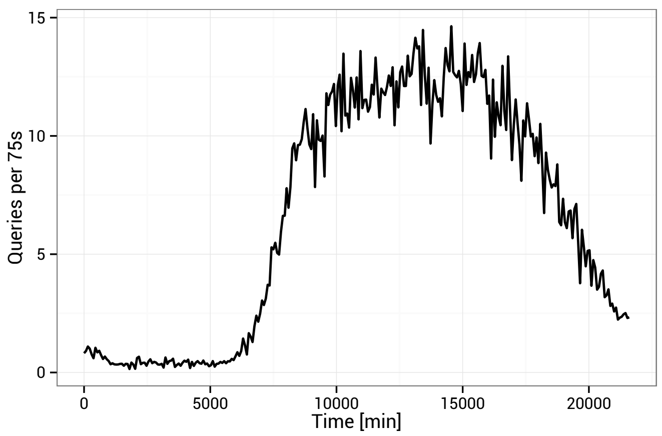


Figure 6.7: Workload trace derived from the service demand of a real-world enterprise application

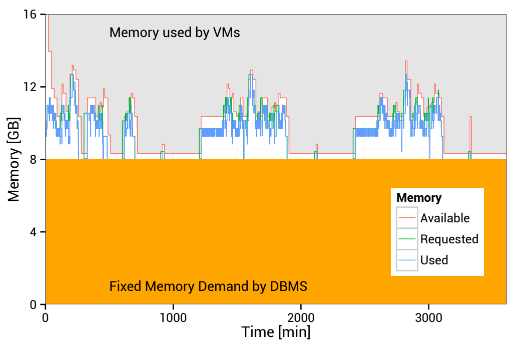


Figure 6.8: Experimental results for the worst-case workload scenario

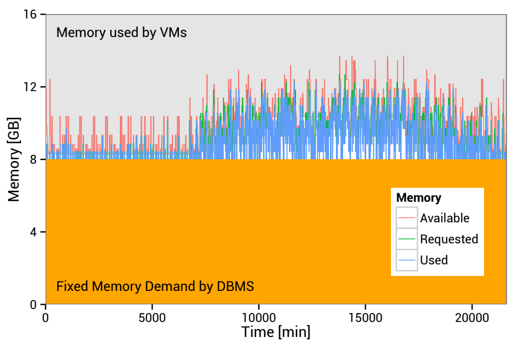


Figure 6.9: Experimental results for the real-world workload scenario

The second workload scenario is derived from log files of a SAP enterprise application used at a large European IT service provider. The log files describe the number of arriving application queries over time (see Appendix E). They were normalized to the maximum number of queries the experimental setup could handle (17 queries in a 75 s interval) as shown in Figure 6.7. The workload specifies the number of queries executed for each interval, with the query type mix (Q1: 70 %, Q2: 20 %, Q3: 10 %). For both workloads, we assume a response time goal of 5 min for all query types and we assume that the maximal throughput of the workload corresponds to the expected arrival rate. This represents the SLAs requirements for our experiments.

Figure 6.8 shows the results of the experiments with the first workload over a runtime of 60 min. Without consolidation, no VMs were running on the database server and all of the queries met their response time goal. The average response time was 103 s with a standard deviation of 80 s. With activated consolidation, the SLAs violations increased by 16 % to 67 out of 414 executed queries, the average response time increased by 43 % to 182 s with a standard deviation of 117 s. In total 89 % of the spare memory could be recycled by executing 84 VM migrations.

Figure 6.9 shows the results of the experiments with the second workload over a runtime of 6 h. Without consolidation, no VMs were running on the database server and there were no SLA violations. The average response time was 14 s with an standard deviation of 6 s. With activated consolidation, there still were no SLA violations, the average response time increased by 41 % to 24 s and the standard deviation increased slightly to 15 s. In total 83 % of the spare memory could be recycled by executing 399 migrations.

6.4 Conclusions

Business servers running database systems are always sized to handle peak query loads without failing their SLAs. In case of main-memory DBMS servers, there is a huge amount of memory which is sized for peak loads. During normal operation a significant amount of memory remains unused. We addressed the question, if this spare memory can be recycled by migrating VMs towards the database server and what the implications are for the DBMS regarding response times and SLAs violations.

The conducted experiments show, that the spare memory can be recycled almost completely. But the database metrics are affected while additional VMs are co-located on the database server. This was an expected behavior as the database server was occupied with lots of additional work. However, the most important metric is the number of SLA violations, because it tells whether the server has enough capacity to run the application within specifications or not. SLAs violations did not change with a continuous query workload without spikes and increased significantly for bursty workloads. This is caused by the time needed to migrate VMs to make room for executing query bursts.

Independent to the workload, the number of executed migrations was astonishingly high. That is a major drawback as migrations cause a huge amount of network traffic and consume CPU cycles. Worse, each migration duplicates the memory demand of a VM during migration as its memory is allocated on the source as well as the destination server. This negatively effects the potential memory savings, especially considering the huge amount of migrations. Recently, extensive performance models for migration have been developed in the cloud computing community [46] and Ye et al. propose to apply resource reservation methods to VM migrations [102].

The number of migrations and SLAs violations could be reduced by using a more advanced query scheduler. In our experiments only a very simple FIFO based query scheduler was used. However, if the query scheduling actually adapted to the available memory, number of migrations as well as SLAs violations could be decreased significantly. One goal is to achieve a memory demand which does not entail spikes and has a low variation. Then, VM allocations need to be adjusted less often, which results in fewer migrations. In presence of memory demand spikes, it may be necessary to sacrifice memory on the database server as buffer capacity to reduce the number of migrations.

The experimental setup showed that it is possible to recycle spare memory resources on a main-memory database server efficiently. By taking precautions, the mentioned problems can be avoided, so that large data centers with lots of servers can substantially reduce the number of active servers and increase their overall utilization.

Part II

Allocation of non-persistent services

Chapter 7

Dynamic VM allocation

Modern data centers are increasingly using virtualization technology and provide VMs for their customers rather than physical servers. Actually, IT service managers worldwide ranked virtualization and server consolidation as one of their top priorities in the recent years [40, 59]. In this work we focus on IaaS as the most basic cloud-service model, in which IT service providers offer servers or VMs as a service to their (internal) customers. VMs can be allocated, deallocated, and moved within seconds using nowadays virtualization and migration technology.

With the adoption of virtualization technology the demand for new physical servers decreased while the demand for VMs has grown considerably. At the same time, server administration costs increased as many VMs need to be managed [42]. This leads to new resource allocation problems, which require decision support and ultimately automation to bring down administration costs and to achieve high energy efficiency. In Part I we addressed the capacity planning in stable environments were VMs are not allocated and deallocated frequently. The literature on dynamic VM allocation in clouds with incoming and outgoing VMs has not yet received much attention. In particular, there is hardly any experimental literature comparing different methods with respect to their energy efficiency. However, experimental tests are important because the many software and hardware components and the various system latencies are difficult to analyze in analytical models or simulations only. Experiments on VM allocation controllers in IaaS clouds are expensive and time consuming which might explain the absence of such studies in the literature. They are important, because the VM allocation controllers can have a substantial impact on the energy efficiency and total cost of IT service provisioning [86].

Let us briefly discuss the state-of-the-practice. Cloud management tools such as OpenStack[1] and Eucalyptus[2] are used in many IaaS cloud environments for VM provisioning. Incoming VMs are placed on servers via simple bin packing heuristics and remain there until they are deallocated. Because such placement controllers do not consider future allocation and deallocation requests, servers might be underutilized and operated at a low energy efficiency.

VM migrations allow to move VMs between servers during runtime. The technology has matured to a state where it is a viable option not only for emergency situations [60], but also for routine VM allocation tasks. At this point, none of the existing cloud management tools uses migration and reallocation in their VM allocation controllers. Rather they rely on the original placement decisions using simple bin packing algorithms.

In this work, we study the energy efficiency of different VM allocation controllers in virtualized data centers. The complexity of nowadays data center infrastructures with the large number of hard- and software components renders analytical models of such infrastructures intractable. Most literature in this area is restricted to simulations only. We present the results of extensive experiments on a data center infrastructure to achieve high external validity of our results. As for Part I, the technology used in our experimental infrastructure closely resembles an IaaS cloud that one would find in small to medium sized corporate environments. The workloads are based on workload data from two large European IT service providers and benchmark applications such as SPECjEnterprise2010[3]. Such experiments are expensive and require an extra infrastructure to run controlled experiments, which can explain why there is a

[1]https://www.openstack.org/
[2]https://www.eucalyptus.com
[3]http://www.spec.org/jEnterprise2010/

lack of such experiments in the literature.

We analyzed different VM allocation controllers, simple *placement controllers* based on bin packing, and advanced ones combining placement and dynamic controllers for server consolidation and high energy efficiency.

There are many possibilities how dynamic controllers can be implemented and combined with placement controllers, and of course one cannot analyze all possible combinations and parameters. In our experiments, we systematically analyzed a large variety of VM allocation controllers in different workload environments and in simulations in a first step. In particular, we compared simple placement controllers as they are regularly used to dynamic controllers, and could show that the latter substantially increase energy efficiency. In a second step, we compared selected placement and dynamic controllers in lab experiments, which allowed us to derive results on various efficiency and quality-of-service metrics with high external validity. Based on our extensive set of experiments, we could identify VM allocation controllers, which perform well in a wide variety of workload environments. The scope and scale of the experiments is beyond what has been reported in the literature, and it provides tangible guidelines for IT service managers.

In the experiments, we found that dynamic controllers have a substantial positive impact on the energy efficiency of a data center. This is not obvious, because VM migrations cause additional workload on the source and target servers. We achieved the highest energy efficiency with placement controllers which computed a dense packing on a low number of servers first. In case of overload, VMs were then migrated and reallocated over time. If the placement decisions were based on the actual demand on a server rather than the reservations for the VMs on a server, the density of the packing could be increased and therefore also the utilization of the servers. Interestingly, the type of bin packing heuristic for the placement controller had little impact on the energy efficiency. Periodic reallocation, however, had substantial impact on the energy efficiency overall.

In Section 7.1, we discuss related literature. Sections 7.2 through 7.5 introduce the experimental setup, while Section 7.6 describes our results. Finally, Section

7.8 provides a summary and conclusions.

Texts in this chapter are based on a previous publication [99].

7.1 Related work

Our research draws on different literature streams. First, we will introduce relevant literature on bin packing problems on a static set of VMs, as this is the fundamental problem for the VM placement. Next we will discuss the literature on placement and dynamic controllers in environments were VMs arrive and depart continuously.

7.1.1 Bin packing

The bin packing problem has been subject of extensive theoretical research. A wide selection of heuristics based on different packing strategies exists [20]. The VM placement problem with a fixed set of VMs can be modeled as a bin packing problem. It can be solved by well known heuristics like Next-Fit, First-Fit, Best-Fit, or many others as evaluated in Part I. Vector bin packing heuristics are an viable option if multiple resources (CPU, RAM) need to be considered or workload profiles are known in advance as shown in Chapter 5.

Coffman [19] analyzed dynamic bin packing where new items arrive and existing ones depart. He proves a competitive ratio that compares dynamic on-line packing heuristics with optimal solutions for the offline problem. The competitive ratio was improved successively, most recently by Wong et al. [100] to a value of $8/3$. Of course, competitive ratios are worst case bounds and average results are typically better than this.

Finally, Ivkovic et al. [43] introduced fully dynamic bin packing where items can be reallocated *before* adding a new item to a bin. Their algorithm is $5/4$-competitive. This theoretical analysis indicates that there are gains from reallocation from the worst-case perspective assumed in the theoretical literature. It is interesting to understand, if reallocation has a positive impact on the average utilization in a data center and what the order of magnitude of these differences is on average in a controlled lab experiment.

7.1.2 Scenarios with a fixed set of VMs

In Part I we addressed stable scenarios where a set of VMs is allocated to a minimal set of servers in order to increase resource utilization. Sometimes VMs can be migrated between servers over time, but the set of VMs is stable as is often the case in enterprise data centers.

Some solutions to these static allocation problems leverage bin packing heuristics, others use integer programming to find optimal or near-optimal solutions [86, 78]. Bobroff et al. [12] predict future resource utilization by autoregressive forecasting models which are used by bin-packing heuristics. pMapper [92] computes a new allocation based on actual resource utilization measurements and triggers iterative VM migrations to change the allocation considering costs entailed with VM migrations. Gmach et al. [34] proposed a fuzzy logic based dynamic controller that load balances servers. Thresholds on memory and CPU are used to detect overloaded servers.

So far, only a few authors evaluated their VM allocation algorithms in the lab. Wood et al. [101] proposed Sandpiper as a dynamic controller for load balancing VMs rather than minimizing total server demand. Migrations are triggered if server overload thresholds are reached. For our experiments, we adapted Sandpaper in one of the treatments such that it minimizes total server demand in addition to load balancing. Sandpiper was evaluated on a hardware infrastructure, but not compared to alternative approaches regarding energy efficiency.

vGreen [25] is another dynamic controller that leverages different controllers depending on the resource considered. VMs on servers with low utilization are moved to other servers heuristically such that some servers can be turned off. vGreen was also evaluated in a small testbed with two servers.

7.1.3 Scenarios with a variable set of VMs

The work discussed so far did not consider on-demand scenarios where VMs arrive and depart over time, as it is the focus of this work. To our best knowledge, only two recent papers have focused on similar problems.

Calcavecchia et al. [15] proposes a two stage VM placement controller called backward speculative planning. Demand risk is a metric characterizing the unfulfilled demand of VMs running on a server. Incoming VMs are placed by a Decreasing-Best-Fit strategy on the server providing the minimal demand risk after placing the VM, assuming that the new VM was fully utilized in the past. VM migrations are only triggered if unfulfilled demand exists and a threshold on the number of migrations over a period is not exceeded. The work focuses on a variant of the First-Fit VM placement and a novel dynamic controller.

Mills et al. [55] simulated placement controllers in an IaaS environment without migrating VMs. For each incoming VM, a cluster and a server are selected in this order. Three cluster selection algorithms and 6 server selection heuristics were evaluated in 32 workload scenarios. Workloads were generated randomly based on a configuration of 6 parameters. Dynamic controllers were not considered by this work.

Both papers analyze VM allocation controllers in fully dynamic scenarios. While these papers are important first steps, we extended these in important ways. First, we analyzed different controllers not only in simulations, but also in lab experiments. This is expensive, but important for the external validity of such experiments. Our workload data used in the simulations and experiments is based on monitoring data of actual enterprise data centers, and VM arrival and departure rates are based on statistics described by Peng et al. [63] based on multiple data centers operated by IBM.

Second, we significantly increased the scale of the studies and analyzed a large number of VM placement and dynamic controllers separately and in combination to take interaction effects into account. This allows us to make recommendations for IT service managers based on a large number of treatment combinations. In particular, we can shed light on the benefits of reallocation in fully dynamic environments. While the number of possible heuristics to allocate VMs dynamically is infinite, we argue that we have considered the most prominent approaches discussed so far as well as a new and promising dynamic controller.

7.2 Experimental setup

In what follows, we describe the experimental setup and the technical infrastructure used for the experiments.

In our infrastructure VMs are created and removed automatically by means of a network service. New VMs are allocated and existing ones are removed continuously. VM lifetimes vary between a couple of minutes and hours. We do not assume prior knowledge about the type or applications running within VMs. An allocation request includes the VM resource reservation and a reference to a disk image used to create it. A *reservation* describes the size of a VM in terms of allocated CPU cores, memory, network, and disk capacity.

For a new VM allocation request, the cloud management tool has to decide on which server the new VM should be placed. This decision is taken by a *placement controller*. Already running VMs might get migrated to another server. Migrations are triggered by *dynamic controllers* that either run in regular intervals or closely monitor the infrastructure and respond to inefficiencies of the current allocation.

In this work, we attempt to analyze a wide variety of implementations for placement and dynamic controllers. The functionality of the controllers under consideration is outlined in the following.

7.3 Placement controllers

Let us first introduce two ways how parameters for placement controllers can be computed. We describe *residual capacity* as the amount of unused resources on a server. It is expressed as a vector. Each component of the vector represents a resource like CPU or memory. There are two types of residual capacity. *Reservation-based residual capacity* subtracts VM resource reservations from a server's total capacity. *Demand-based residual capacity* is the measured amount of free resources on a server. Both can be used for placement controllers.

Interestingly, all available cloud management tools that we are aware of (including OpenStack and Eucalyptus) use a reservation-based allocation, which guarantees each VM its resource reservation. Many VMs are not utilized to 100 % of the capacity that was reserved leading to underutilized servers. Demand-based allocation leverages information about the actual VM resource utilization and increases server utilization by overbooking. However, they can only be used in conjunction with dynamic controllers as otherwise servers might get overloaded. Dynamic controllers mitigate overloaded servers using VM migrations.

Any-Fit placement controllers including First-Fit, Best-Fit, Worst-Fit, and Next-Fit assign an arriving VM* to a server [20]. These are regularly used in cloud management software and based on established and simple bin packing algorithms [75]: First-Fit-Decreasing iterates over the decreasingly sorted server list by their load and picks the first one that fits the VM* reservation. Best-Fit computes the delta of residual capacity and VM* reservation vector norms and picks the server with minimal difference that fulfills the reservation. Worst-Fit works the same way but selects the servers with the maximum vector norm difference. Next-Fit holds a pointer on a server. Each incoming VM is placed on this server as long as its residual capacity allows for the VM*'s reservation. Otherwise, the pointer is moved to the next server in a round-robin fashion until a feasible server is found. All Any-Fit implementations activate an empty server if no viable server could be found.

Apart from these standard algorithms, there have been specific proposals for

VM placement in the literature. Dot-Product [90] is a First-Fit-Decreasing approach where server weight is calculated by the dot product of server residual capacity, s^{-1}, and the reservation r_{VM}^{-1} for VM^* as: $s^{-1} \cdot r_{VM}^{-1}$. Similarly, cosine is also a First-Fit-Decreasing algorithm [52] that uses the cosine as a weight function: $cos(s^{-1}, r_{VM}^{-1})$. L2 [90] leverages First-Fit-Increasing with the difference of the vector norms as a weight $\|s^{-1}\| - \|r_{VM}^{-1}\|$.

7.4 Dynamic controllers

Dynamic controllers are executed regularly and trigger VM migrations in order to reoptimize the allocation. We found two approaches in the literature to compute a schedule of reallocations over time. DSAP+ describes an optimization model to compute an optimal schedule of reallocations over time and it was introduced by [86], [78], and in Chapter 4. Alternatively, KMControl (see Chapter 2) and TControl are two heuristics based on the Sandpiper system as originally proposed by [101]. In the following, we will briefly describe these controllers.

7.4.1 DSAP+

The DSAP+ controller gets executed every 60 s. Each time it recomputes the allocation of VMs to servers according to the optimization model described by Equation 7.1. The fundamental DSAP+ optimization model was discussed in Chapter 4. We modified it to cover only one subsequent allocation period that gets parametrized with the prevailing VM allocation.

Suppose we are given a set of servers $i \in I$ and VMs $j \in J$. A server's size is denoted by s_i describing its resource capacity, e.g. CPU units or available memory. The total planning horizon is divided into two discrete periods $t \in \{1, 2\}$, the current one x_{ij1} and the upcoming one x_{ij2}. Values for x_{ij1} are passed as a parameter to the model. $y_i \in \{0, 1\}$ tells whether a server i is active in period 2. u_j describes the expected utilization of VM j during period

2. The allocation matrix x_{ijt} of period t indicates whether VM j is assigned to server i. Migrations of VMs from period 1 to 2 are indicated by slack variables z_{ij}^{-} for outgoing ones. The objective function minimized the sum of total server operation costs o^{srv} and migration costs o_j^{mig}.

$$
\begin{aligned}
&\min \sum_{i=1}^{I} \left(o^{\text{srv}} y_{i2} + \sum_{j}^{J} o_j^{\text{mig}} z_{ij}^{-} \right) && && \\
&\text{s.t.} && && \\
&\sum_{i=1}^{I} x_{ij2} && = 1, && \forall j \in J \\
&\sum_{j=1}^{J} u_j x_{ij2} && \leq s_i y_i, && \forall i \in I \\
&- x_{ij1} + x_{ij2} - z_{ij}^{-} && \leq 0 && \forall i \in I, \forall j \in J \\
&y_i, x_{ijt}, z_{ij}^{-} \in \{0,1\}, && && \forall i \in I, \forall j \in J, \forall t \in \{1,2\}
\end{aligned}
\tag{7.1}
$$

After each execution, the new allocation x_{ij2} is implemented by triggering migrations. A migration scheduler ensures that each server is running only one migration in parallel, either an outgoing or an incoming one. As many migrations as possible are executed in parallel without overloading servers or their network interfaces.

For parametrization, server operation costs and migration costs are estimated by execution period duration and average migration duration. In our case, execution period duration was 60 s while migrations took 25 s on average (see Chapter 2) which provides values for parameters $o^{srv} = 60$ and $o_i^{mig} = 25$ in the objective function.

7.4.2 KMControl and TControl

In this study we leverage two variants (KMControl and TControl) of the proactive controller used in Chapter 2 that itself is based on Sandpiper [101]. Each gets executed periodically every 5 min. There are three phases: 1) Detect and mark overloaded and underloaded servers 2) Compute server and VM load rankings 3) Mitigate overloaded and underloaded servers by triggering VM migrations.

KMControl and TControl differ in phase one. KMControl checks if M out of the last K server CPU utilization measurements are above an threshold T_o or below T_u to mark a server as overloaded or underloaded. TControl is based on a single-sided t-test that compares the mean of the M recent utilization measurements starting at time t against T_o and T_u: $T_o \leq \frac{1}{M} \sum \{u_{t-(M-1)}, .., u_t\} \leq T_u$. If H_0 is rejected at a p-level of 0.001, a significant difference is assumed and a server is either marked as overloaded or underloaded.

Phase two computes a volume VOL (Equation 7.2) and a volume-size ratio VSR (Equation 7.3) for each server and VM according to [101] with u^r denoting the utilization of resource r. Both computations are performed for physical servers and VMs. Servers are sorted by VOL in decreasing order so that servers with a high resource utilization are found at the top.

$$VOL = \frac{1}{\prod\limits_{\forall r}(1 - u^r)} \tag{7.2}$$

$$VSR = \frac{VOL}{c^{mem}} \tag{7.3}$$

Phase three triggers migrations such that underloaded servers are emptied and overloaded ones are relieved. VMs running on an overloaded or underloaded server are sorted by their VSR in an decreasing order. Here, c^{mem} describes the memory capacity of a server or VM. VM migration overhead depends on the combination of memory size and server utilization. VSR puts memory size into perspective of utilization. To reduce migration costs, VMs with a low memory size compared to their utilization are considered first for migration. VMs on underloaded servers are preferably migrated to servers at the head while VMs on overloaded servers are migrated towards the tail of the sorted server list.

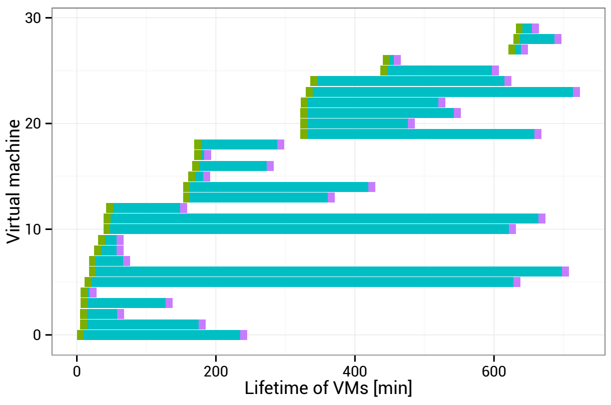


Figure 7.1: Sample of a VM arrival and departure schedule

7.5 Experimental design

For our experiments we generated schedules of VM arrivals and departures that we could evaluate with different controllers. This allows a fair comparison of different combinations of placement and dynamic controllers. A schedule as shown in Figure 7.1 is generated based on four random variables:

- **Lifetime** of a VM (width of the bars in Figure 7.1)
- **Inter-arrival time** of the VMs
- **VM launches** describes the total number of arriving VMs (total number of bars in Figure 7.1)
- **VM sizes** describes the VM reservations

Not all possible schedules could be evaluated, considering multiple levels for each variable. In order to evaluate only relevant schedule configurations we analyzed which variables have the most impact to the performance of placement controllers. A 2^k fully factorial design was used to address this question. Table 7.1 summarizes factors and levels. For each possible schedule configuration, 25 random schedule instances were generated.

Factor	Low	High
Lifetime (h)	1	6
Inter-arrival time (min)	5	20
VM launches	400	500
VM sizes	2^x	x

Table 7.1: Factors and levels for the schedule generation. Lifetime and inter-arrival time are chosen based on a negative exponential distribution. For each configuration 25 schedule instances were generated randomly.

Each schedule was evaluated with 9 placement controllers: First-Fit, Best-Fit, Worst-Fit, Dot-Product with a demand- and reservation-based implementation. A random controller was used as a control group. In total $16 \cdot 25 \cdot 9 = 3600$ simulations were conducted.

For each simulation the allocation density of the controller was calculated based on average and peak server demand during the simulation and a lower bound estimate. A dedicated ANOVA analysis for each controller found lifetime and inter-arrival time as well as their interaction effect significant in all cases, with levels of $p < 0.001$. In rare cases, the factor launches was significant, with levels of $p < 0.01$. Q-Q plots and residual plots showed no abnormalities. More detail on this analysis can be found in Section E.3.

A single experiment takes approximately 13 h to 15 h. Schedules used in experiments vary factors lifetime and inter-arrival time of VMs as suggested before. Inter-arrival times are taken from Peng et al. [63], who published cumulative density functions (CDFs) for inter-arrival times of VMs. Data originated from real-world measurements in enterprise data centers operated by IBM. Our schedules leverage two CDF functions, one with short (A1) and one with longer inter-arrival times (A2).

Lifetimes are based on two CDFs from Peng et al. as well. L1 is a CDF of VM life-times over all data centers, L2 is a mixture of CDFs for unpopular, average, and popular VMs with probabilities $p = (0.2, 0.6, 0.2)$. Both, VM inter-arrival- and life-times are scaled by factor 30 to keep the total time of an experiment below 15 h. Table 7.2 provides an overview of the schedule configurations in our experiments.

Schedule Config.	Arrival Time	Lifetime
1	A1	L1
2	A1	L2
3	A2	L1
4	A2	L2

Table 7.2: Five schedule instances were generated for each configuration shown in the table

Five schedule instances were generated for each schedule configuration shown in Table 7.2. A schedule runs up to 20 VMs in parallel. Three VM sizes were used: Small VMs (S) were configured with 1 vCPU core and 2 GB of memory. Medium sized VMs (M) had 2 vCPU cores and 4 GB of memory. Large ones (L) were assigned 3 vCPU cores and 6 GB of memory. VM sizes were picked based on probabilities $(S, M, L) = (0.6, 0.3, 0.1)$ for each VM in a schedule.

A detailed description of the experimental testbed can be found in Chapter A. Rain was configured to simulate a varying amount of users over time. This user demand was specified by a set of 32 time series that are similar to the ones used in Part I and describe the workload for a given time in values $0 \leq l \leq 1$. Depending on the VM size, the time series were multiplied with 100 (S), 150 (M), and 200 (L) users. The resulting time series describes the number of users to simulate by Rain over time on a single target VM.

7.6 Simulation results

The experiments compare different combinations of placement and dynamic controllers. Due to the large number of possible controller combinations one can analyze, we first conducted simulations and ranked them based on core metrics: migrations, average, and maximum server demand. Subsequent experiments in the lab were then conducted on the most promising controllers only.

Simulations were conducted for 20 schedule instances described in Section 7.5 and all combinations of 11 placement and 4 dynamic controllers, including scenarios without dynamic controllers. A more detailed description of the simulation framework used can be found in Chapter A.2.

Controllers are named by an approach similar to the Kendall notation with three variables separated by a slash, e.g. D/FF/KM. The first element declares if a demand (D) or reservation (R) based placement controller was used. The placement controller used is described in the second element with FF=First-Fit, BF=Best-Fit, WF= Worst-Fit, RD = Random, DP = Dot-Product, L2 = L2. The dynamic controller is described in the last element with KM = KMControl, TC = TControl, DP = DSAP+, and — if no dynamic controller was used.

Based on the simulations, controllers were ranked using the mean reciprocal rank (MRR) metric. An MRR rank is computed by Equation 7.4, larger values indicate better rankings.

$$\mathrm{MRR}(c) = \frac{1}{|P| \cdot |Q|} \sum_{p \in P} \sum_{q \in Q} \frac{1}{R_{cpq}} \tag{7.4}$$

For each controller $c \in C$ and schedule instance $q \in Q$, three ranking values $p \in P$ exist: migrations (MG), average ($\overline{\mathrm{SD}}$), and maximum server demand ($\lceil \mathrm{SD} \rceil$).

Rankings R_{cpq} are calculated for each metric and schedule separately. For each schedule instance, controllers are sorted in increasing order by a metric (MG, $\overline{\mathrm{SD}}$, $\lceil \mathrm{SD} \rceil$). A controller's list position gives its ranking. In total, each controller is assigned $|P| \cdot |Q|$ rankings.

Controllers are sorted by the MRR ranking over all metrics in Table 7.3. Some combined controllers using placement and dynamic controllers seem to be equally good than controllers without dynamic strategy, still pure static controllers outperformed all dynamic controllers by the MRR ranking. Obviously, placement-only controllers do not trigger VM migrations. This is an advantage in the combined ranking as they get the best possible score on the migrations metric. Despite this advantage, they often performed worse than

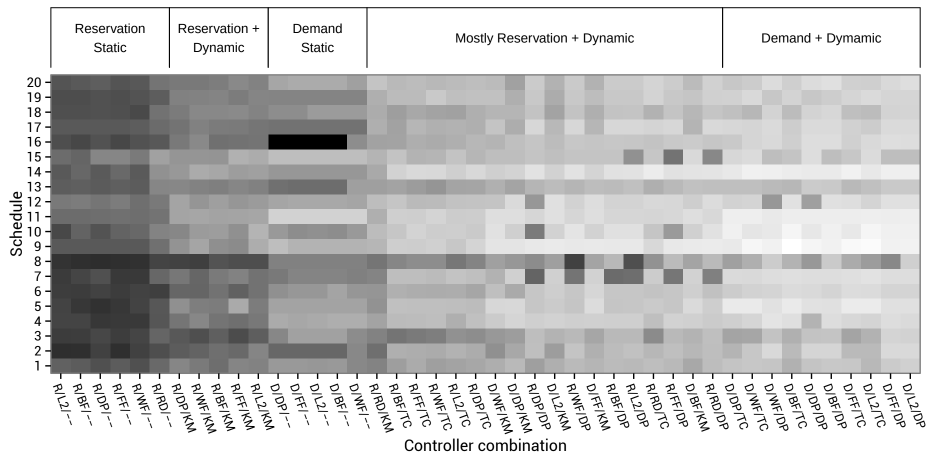


Figure 7.2: Heatmap of **average server demand** over all controllers and schedule instances

combinations of placement and dynamic controllers for the MRR rankings $\overline{\text{SD}}$ and $\lceil\text{SD}\rceil$.

Figure 7.2 shows a heatmap of the average server demand for each controller and schedule. Controllers are sorted by their MRR ranking on the average server demand only, not considering other metrics as for Table 7.3. Darker areas indicate high values and bright ones low values.

In the heatmap the controllers can be clustered following the type of controller combination. Controllers without dynamic strategy performed worst and are found on the left hand side (Reservation Static). Demand-based controllers outperformed reservation-based ones. Controllers leveraging some kind of dynamic strategy usually delivered a higher allocation density. Again, combined controllers that leverage demand-based placement controllers outperformed reservation-based ones (Reservation + Dynamic and Mostly Reservation + Dynamic vs. Demand + Dynamic).

Reservation-based controllers performed worst because they are most conservative. Reservations do not reflect the actual resource demand of a VM. Adding a dynamic controller overall improved allocation density. Using a demand-based instead of a reservation-based controller takes advantage of the actual

	MRR Rankings				Average Simulation Results		
Controller	$\overline{\text{MRR}}$	$\overline{\text{SD}}$	$\lceil\text{SD}\rceil$	$\overline{\text{MG}}$	$\overline{\text{SD}}$	$\lceil\text{SD}\rceil$	$\overline{\text{MG}}$
D/WF/--	0.52	0.04	0.52	1.00	3.33 (0.37)	5.30 (0.57)	0.00 (0.00)
D/BF/--	0.48	0.03	0.45	0.95	3.34 (0.45)	5.42 (0.61)	0.00 (0.00)
D/L2/--	0.48	0.03	0.44	0.95	3.36 (0.44)	5.47 (0.61)	0.00 (0.00)
D/FF/--	0.47	0.03	0.42	0.95	3.37 (0.44)	5.53 (0.51)	0.00 (0.00)
D/DP/--	0.47	0.03	0.41	0.95	3.39 (0.46)	5.58 (0.51)	0.00 (0.00)
R/RD/--	0.46	0.03	0.36	1.00	4.10 (0.43)	6.00 (0.00)	0.00 (0.00)
R/WF/--	0.46	0.03	0.36	1.00	4.52 (0.35)	6.00 (0.00)	0.00 (0.00)
R/FF/--	0.46	0.03	0.36	1.00	4.53 (0.45)	6.00 (0.00)	0.00 (0.00)
R/DP/--	0.46	0.03	0.36	1.00	4.52 (0.39)	6.00 (0.00)	0.00 (0.00)
R/BF/--	0.46	0.03	0.36	1.00	4.54 (0.38)	6.00 (0.00)	0.00 (0.00)
R/L2/--	0.46	0.03	0.36	1.00	4.57 (0.36)	6.00 (0.00)	0.00 (0.00)
D/L2/DP	0.44	0.46	0.73	0.12	2.31 (0.20)	4.65 (0.49)	13.05 (3.35)
D/BF/TC	0.42	0.26	0.85	0.14	2.44 (0.36)	4.35 (0.81)	10.75 (2.43)
D/L2/TC	0.42	0.31	0.80	0.14	2.43 (0.37)	4.45 (0.69)	10.80 (2.84)
D/BF/KM	0.40	0.09	0.79	0.33	2.60 (0.30)	4.50 (0.69)	5.30 (2.11)
D/FF/TC	0.40	0.29	0.78	0.15	2.44 (0.35)	4.50 (0.69)	10.40 (2.39)
D/WF/TC	0.39	0.20	0.82	0.14	2.43 (0.32)	4.40 (0.68)	10.60 (2.44)
D/L2/KM	0.39	0.07	0.71	0.39	2.70 (0.34)	4.70 (0.66)	4.60 (1.70)
D/FF/KM	0.38	0.08	0.69	0.36	2.64 (0.31)	4.70 (0.57)	4.75 (1.68)
D/FF/DP	0.38	0.31	0.68	0.13	2.37 (0.35)	4.75 (0.44)	12.45 (3.52)
D/DP/KM	0.36	0.06	0.66	0.38	2.76 (0.37)	4.80 (0.70)	4.75 (1.71)
D/BF/DP	0.35	0.28	0.66	0.13	2.34 (0.22)	4.80 (0.52)	13.00 (3.43)
D/DP/TC	0.35	0.12	0.80	0.14	2.49 (0.35)	4.45 (0.69)	10.70 (2.27)
D/WF/DP	0.35	0.26	0.67	0.14	2.42 (0.33)	4.85 (0.75)	12.75 (4.34)
D/DP/DP	0.35	0.27	0.65	0.14	2.45 (0.37)	4.85 (0.75)	12.45 (3.83)
D/WF/KM	0.35	0.05	0.66	0.34	2.75 (0.36)	4.80 (0.52)	5.05 (1.70)
R/RD/TC	0.24	0.08	0.56	0.08	2.67 (0.36)	5.20 (0.52)	17.85 (3.31)
R/RD/DP	0.21	0.10	0.44	0.10	2.63 (0.42)	5.70 (0.47)	15.70 (3.87)
R/L2/DP	0.21	0.08	0.46	0.10	2.75 (0.61)	5.70 (0.47)	18.95 (5.84)
R/WF/DP	0.21	0.07	0.46	0.09	2.72 (0.62)	5.70 (0.47)	19.80 (5.05)
R/FF/DP	0.21	0.08	0.46	0.08	2.74 (0.48)	5.70 (0.47)	19.60 (5.07)
R/RD/KM	0.21	0.04	0.43	0.15	3.22 (0.40)	5.75 (0.44)	10.60 (2.58)
R/BF/DP	0.20	0.08	0.46	0.07	2.64 (0.40)	5.70 (0.47)	20.85 (3.87)
R/DP/DP	0.20	0.07	0.46	0.08	2.82 (0.54)	5.70 (0.47)	19.05 (4.87)
R/WF/TC	0.19	0.04	0.47	0.06	2.91 (0.38)	5.70 (0.47)	23.95 (2.95)
R/WF/KM	0.18	0.03	0.37	0.13	3.74 (0.48)	5.95 (0.22)	11.40 (3.12)
R/FF/TC	0.18	0.04	0.42	0.06	2.90 (0.36)	5.70 (0.47)	23.75 (4.54)
R/DP/KM	0.18	0.03	0.36	0.14	3.78 (0.42)	6.00 (0.00)	11.55 (2.91)
R/L2/KM	0.18	0.03	0.36	0.14	3.65 (0.44)	6.00 (0.00)	11.90 (2.88)
R/L2/TC	0.17	0.04	0.42	0.06	2.91 (0.35)	5.75 (0.44)	23.10 (3.86)
R/FF/KM	0.17	0.03	0.36	0.13	3.71 (0.46)	6.00 (0.00)	11.40 (2.68)
R/BF/TC	0.17	0.04	0.42	0.06	2.94 (0.39)	5.75 (0.44)	24.25 (3.48)
R/DP/TC	0.17	0.04	0.41	0.06	2.87 (0.33)	5.80 (0.41)	24.55 (4.12)
R/BF/KM	0.17	0.03	0.36	0.12	3.73 (0.37)	6.00 (0.00)	12.25 (2.17)

Table 7.3: $\overline{\text{MRR}}$ – average MRR ranking of all metric MRR rankings (migrations, average, and maximum server demand), $\overline{\text{SD}}$ – average server demand, $\lceil\text{SD}\rceil$ – maximum server demand, $\overline{\text{MG}}$ – average number of VM migrations

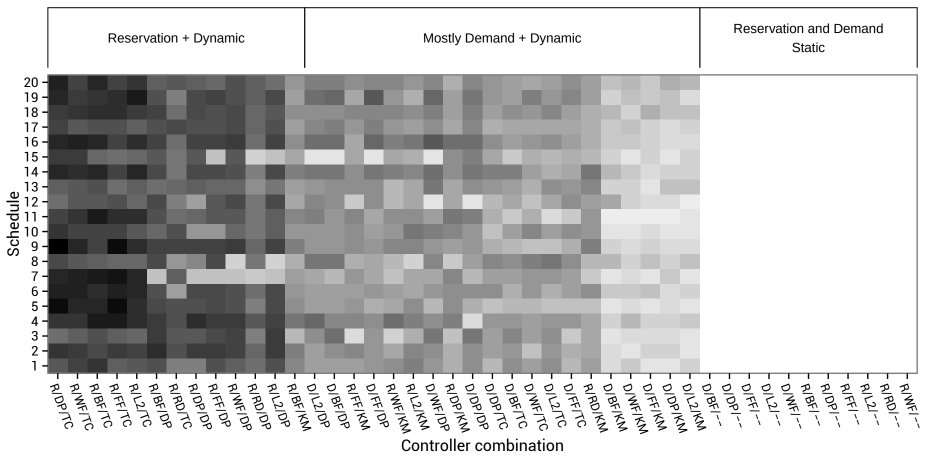


Figure 7.3: Heatmap of **migrations** over all controllers and schedule instances

resource demand and achieves a denser allocation in the first place. Again, adding a dynamic controller improved allocation density. This combination performed even slightly better as a combination of reservation-based and dynamic controllers. Reasons are, demand-based controllers achieve a denser allocation right after placement and they achieve a denser allocation faster than reservation-based ones due to less VM migrations.

With respect to VM migrations, controllers without dynamic strategy did not trigger any migrations and performed best (to be found on the right in Figure 7.3). Demand-based controllers consistently triggered less migrations than reservation-based ones. Demand-based controller combinations outperformed reservation-based ones for the same reason as before. Leveraging VM demands leads to a denser allocation in the first place. Fewer or even zero migrations are required to establish a dense allocation after allocating a new VM.

Table 7.4 summarizes the average server demand and number of VM migrations clustered by the controller type. This provides additional evidence that demand-based placement controllers lead to a superior VM allocation in contrast to reservation-based ones, and that combining them with dynamic controllers reduces the number of migrations substantially in contrast to reservation based placement controller combinations.

Cluster	$\overline{\text{SD}}$	σSD	ΔSD	$\overline{\text{MG}}$	σMG	ΔMG
Reservation Static	4.46	0.42	2.03	0.00	0.00	0.00
Demand Static	3.36	0.42	1.70	0.00	0.00	0.00
Reservation + Dynamic	3.07	0.60	2.80	17.81	6.27	28.00
Demand + Dynamic	2.51	0.35	1.81	9.43	4.30	18.00

Table 7.4: Statistics for results clustered by the controller type. $\overline{\text{SD}}$ and $\overline{\text{MG}}$ – average server demand and migrations, σSD and σMig – standard deviation for average server demand and VM migrations, ΔSD and ΔMig – difference between min and max values of average server demand and migrations

Overall, simulations suggest that combinations of demand-based placement with dynamic controllers are most efficient.

7.7 Experimental results

We performed complementary lab experiments in order to understand if the main results of the simulations carry over to a real environment. In addition, the lab experiments provide information about response times and SLA violations of each controller, which cannot be obtained properly by simulations. Experiments were conducted with the following controller combinations:

- Good performing controllers according to simulations considering all MRR rankings separately
  - First-Fit-Demand with KMControl (D/FF/KM)
  - L2-Demand with KMControl (D/L2/KM)
  - Worst-Fit-Demand without dynamic controller (D/WF/–)
- Poor performing controllers according to simulations considering all MRR rankings separately
  - Worst-Fit with TControl (R/WF/TC)
  - Worst-Fit with KMControl (R/WF/KM)

Controller combinations for experiments were chosen to cover a set of good and poor performing controllers based on the MRR ranking shown in Table 7.3. We selected D/FF/KM and D/L2/KM as they performed well for average, maximum server demand, and migrations. R/WF/TC and R/WF/KM represent poor performing controllers. In each case two controllers were picked to see whether they perform similarly well in experiments as suggested by simulations. Considering all metrics and data center requirements there is no clear winner. In addition D/WF/– was tested because it performed best according to the global MRR ranking if migrations are a limiting factor and average server demand is less of a concern.

Experimental results can be found in Table 7.5. For each controller and schedule configuration, there is one result line. It describes the average experimental results over 5 schedule instances of one schedule configuration. Differences between min and max results are reported in square brackets while variances are reported in parenthesis.

For migrations, we found that combinations of reservation-based placement and dynamic controllers triggered more migrations than ones using demand-based placement controllers. This confirms simulation results and can be explained by the higher resource demands of reservation-based controllers. Demand-based controller combinations consistently triggered more migrations than pure static controllers and less than reservation-based combinations.

Service quality could be maintained by almost all controller combinations except D/L2/KM which fell below a desired level of 99 % service quality for all schedule configurations. For some schedule configurations controller combinations R/WF/TC and D/WF/– also fell below a service quality of 99 %.

Average server demand showed that D/FF/KM delivered the best performance and consistently required the least number of servers while maintaining the desired service level. Overall R/WF/KM had the highest average server demand and the highest peak server demand. These results are in line with simulation results. All other controllers can be found between D/FF/KM and R/WF/KM.

	Control	Sched	$\overline{\text{SD}}$	$\lceil\text{SD}\rceil$	$\overline{\text{CPU}}$	$\overline{\text{MEM}}$	$\overline{\text{RT}}$	$\lceil\text{RT}\rceil$	O	$\overline{\text{MG}}$	SQ
+	D/FF/KM	20000	2.53 (0.17)	3.67 (0.58)	42	47	654.50 (49.86)	44847	2460676	07 [05/08]	99.12 (0.273)
+	D/L2/KM	20000	2.74 (0.27)	4.25 (0.96)	42	43	900.19 (187.90)	50262	2399470	06 [05/08]	95.52 (1.360)
+	D/WF/--	20000	3.15 (0.25)	4.75 (0.96)	33	39	617.55 (26.44)	23045	2576691	00 [00/00]	99.60 (0.100)
-	R/WF/KM	20000	3.69 (0.17)	6.00 (0.00)	32	34	637.24 (32.40)	36326	2634552	20 [16/22]	99.61 (0.049)
-	R/WF/TC	20000	2.97 (0.27)	5.50 (0.53)	40	41	679.35 (87.81)	61091	2555583	26 [25/27]	97.70 (0.814)
+	D/FF/KM	20100	2.41 (0.39)	4.00 (0.82)	41	48	627.23 (52.04)	32832	2528618	08 [06/09]	99.48 (0.095)
+	D/L2/KM	20100	2.88 (0.51)	4.50 (0.58)	41	41	1427.40 (518.85)	61127	2210333	06 [04/07]	91.66 (2.033)
+	D/WF/--	20100	3.38 (0.41)	5.25 (0.50)	28	35	616.68 (26.65)	21060	2535460	00 [00/00]	99.49 (0.139)
-	R/WF/KM	20100	3.60 (0.66)	6.00 (0.00)	30	33	615.77 (50.38)	34873	2535183	19 [16/21]	99.50 (0.121)
-	R/WF/TC	20100	2.77 (0.48)	5.75 (0.46)	39	42	641.82 (39.44)	46271	2522725	27 [24/29]	99.18 (0.200)
+	D/FF/KM	20200	2.29 (0.22)	4.50 (0.58)	38	43	662.26 (61.48)	44190	1923776	07 [04/09]	98.77 (0.316)
+	D/L2/KM	20200	2.34 (0.29)	4.25 (0.50)	42	43	1578.25 (611.89)	204359	1672268	06 [04/09]	94.58 (1.515)
+	D/WF/--	20200	2.97 (0.45)	4.75 (0.50)	28	35	635.31 (59.12)	28491	1932057	00 [00/00]	99.21 (0.204)
-	R/WF/KM	20200	3.21 (0.26)	6.00 (0.00)	30	31	639.47 (31.31)	40645	1931438	20 [17/24]	99.16 (0.217)
-	R/WF/TC	20200	2.63 (0.16)	5.43 (0.53)	37	37	674.82 (50.44)	58384	1849433	27 [23/33]	98.68 (0.274)
+	D/FF/KM	20300	2.38 (0.14)	4.00 (0.00)	41	48	670.62 (33.76)	59608	2365682	09 [08/10]	99.23 (0.136)
+	D/L2/KM	20300	2.68 (0.07)	4.33 (0.58)	41	45	1091.45 (152.11)	82731	2270676	07 [07/08]	97.30 (0.284)
+	D/WF/--	20300	3.29 (0.16)	5.00 (0.00)	28	36	644.83 (40.58)	31294	2383054	00 [00/00]	91.90 (3.625)
-	R/WF/KM	20300	3.62 (0.15)	6.00 (0.00)	30	32	648.37 (25.09)	48188	2370031	19 [15/20]	99.38 (0.041)
-	R/WF/TC	20300	2.78 (0.15)	5.25 (0.46)	39	42	686.48 (34.21)	46496	2360173	26 [24/29]	96.80 (0.928)

Table 7.5: Experimental results on placement and dynamic controllers.

(+/-) – based on simulation results, either a poor or good performing controller. Control – controller combination with notation [(D)emand- or (R)eservation-based]/[placement controller]/[dynamic controller], Sched – VM allocation/deallocation schedule configuration, $\overline{\text{SD}}$ – average server demand, $\lceil\text{SD}\rceil$ – maximum server demand, $\overline{\text{CPU}}$ [%] – average CPU load, $\overline{\text{MEM}}$ [%] – average memory load, $\overline{\text{RT}}$ [ms] – average response time, $\lceil\text{RT}\rceil$ [ms] – maximum response time, O – total number of operations, $\overline{\text{MG}}$ – average VM migrations, SQ [%] – service quality (number of requests with a response time greater 3 s plus failed requests vs. all requests)

Server CPU utilization varied between 20 % and 45 % depending on the controller configuration and remained well below 70 %. Average memory as a second constraining resource remained below a 50 % utilization level for all controller combinations. A reason for this low average CPU utilization was that most VMs were only utilized below 30 % on average. In order to increase overall server CPU utilization many VMs would have to be allocated to a single server. However, this was not possible due to a fixed memory limit for all VMs on a server, which was set to 85 % of its total capacity to keep enough space for the hypervisor as shown in Figure 7.4. Server utilization increased

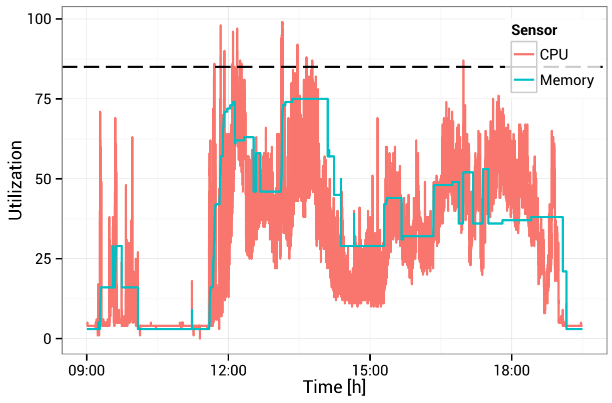


Figure 7.4: Server CPU and memory utilization for one experiment

to 90 % and above during times with higher workload on the VMs.

CPU utilization will also be low if there are too few VMs to fully utilize a single or all active servers. For example an average CPU utilization of 50 % is caused if two servers are active, one running at 80 % and the second one at 20 % utilization. Average utilization cannot be increased because resource limitations prevent a migration of the VM on the second server to the first one, and there are not enough VMs available.

Average response time of operations was slightly sensitive to denser allocations. For the two controllers D/FF/KM, R/WF/TC, and D/L2/KM that produced the lowest average server demand, average response time was slightly increased compared to other controllers. However, the difference is not significant due to the high variance in the response times. Interestingly, D/L2/KM yielded the highest response times and worst service quality while delivering low server demands and maximum server demands without triggering too much migrations.

Table 7.6 summarizes experimental results over all schedules for each controller. While the numbers are slightly different, the ranking of controllers is in line with the simulation results. Average server demand was generally a bit lower as predicted by simulations while more VM migrations were triggered. Demand-based controllers always triggered less migrations than reservation-

Cluster	$\overline{\text{SD}}$	σSD	ΔSD	$\overline{\text{MG}}$	σMG	ΔMG
D/FF/KM	2.40	0.24	0.95	7.60	1.68	6
D/L2/KM	2.66	0.37	1.32	6.07	1.53	5
R/WF/TC	2.79	0.31	1.32	26.42	2.25	10
D/WF/--	3.20	0.34	1.37	0.00	0.00	0
R/WF/KM	3.54	0.38	1.53	19.59	2.43	9

Table 7.6: Statistics over all experiments sorted by MRR ranking in experiments. $\overline{\text{SD}}$ – average server demand, σSD – standard deviation of server demand, ΔSD – difference between the min and max server demand. The same statistics apply for VM migrations MG.

based controllers as for simulations.

We found that average results for R/WF/KM and D/WF/– may not always be in line with simulation results. Due to simulations, R/WF/KM sometimes produced a lower average server demand then D/WF/–. We found, that D/WF/– achieves a very dense allocation if inter-arrival times were high. The setup procedure used to initialize a new VM produces a very low CPU utilization on the servers. In this case, the demand based controller is only limited by the VM memory reservations, resulting a in a very dense allocation. This behavior increases the risk of service quality degradation as seen for schedule configuration 20300 where the demand-based controller required less servers but only achieved a service quality of 92 %.

In simulations we found that the variance on the CPU measurements has a strong effect on the KMControl controller's performance. This is not the case for the TControl controller as it's t-test takes the noise variance into account. If parameters $(K,\ M)$ of KMControl are not properly tuned, the controller reacts faster and more aggressive, leading to a lower average server demand and service quality with an increased number of migrations.

7.8 Conclusions

Much research on VM allocation in virtualized data centers has focused on efficient heuristics for the placement controllers. Typically, bin packing heuristics are used in wide-spread cloud management tools like OpenStack or Eucalyptus. We could not find substantial differences in the energy efficiency of different bin packing heuristics in our placement controllers.

However, there were substantial differences in the energy efficiency if additional dynamic controllers were used. There was no substantial difference among the types of dynamic controllers used, but whether it was used or not had a considerable impact on the average server demand. Surprisingly, reallocation has not been a concern in the related literature so far, nor is it used in cloud management tools used in industry practice. The result is in line with theoretical work on fully dynamic bin packing, which leads to lower competitive ratios than dynamic bin packing.

In addition, the parameters used for the placement controllers have an impact on the average server demand. Both, simulation and experimental results indicate that a controller should aim for a dense allocation from the start for high energy efficiency. This is mainly a result of the parameters used for bin packing. Demand-based placement controllers take the actual demand on a server into account and not the reserved capacity. This allows for a much denser packing and higher utilization. However, demand-based placement controllers need to be used in conjunction with dynamic controllers to avoid overloads.

Nowadays, reservation-based placement controllers are state-of-the-practice, which is probably due to risk considerations of IT service managers. Our study shows that combinations of demand-based placement controllers with dynamic controllers actually lead to fewer migrations than reservation-based placement controllers and a lower server demand at the same time, while maintaining service quality.

Overall, demand-based placement controllers in combination with a dynamic controller appear to be the most energy-efficient solution. Simulations indicate significant savings in average server demand of about 20 % to 30 % compared to placement-only allocation controllers.

The cost of migrations is always an issue. Not only do they lead to higher response times, but they can lead to congestion in the network. In our analysis, we assumed that the network connections within a server cluster are such that they do not become a bottleneck. This can be an issue in large-scale data centers with many migrations and additional network traffic, however. We leave migration scheduling for such scenarios as a topic for future research.

Part III

Appendix

Appendix A

IaaS cloud testbed for research experiments

Texts in this chapter are based on a previous publication [98].

A.1 Hardware infrastructure

All experiments except the ones of Chapter 6 were conducted using an experimental testbed infrastructure that closely resembles the architecture one would find in a private IaaS cloud environment. The overall architecture is shown in Figure A.1.

The testbed consists of six identical servers and many VMs of different sizes. The concrete number of VMs with their resource reservation depends on the experiment and is described in the corresponding chapters. Fedora Linux 16 is used as operating system with Linux KVM as hypervisor. Each server is equipped with a single Intel Quad CPU Q9550 2.66 GHz, 16 GB memory, a single 10 k disk and four 1 Gbit network interfaces.

The VM disk files are located on two separate NFS storage servers as QCOW2 files. The first one is equipped with an Intel Xeon E5405 CPU, 16 GB memory

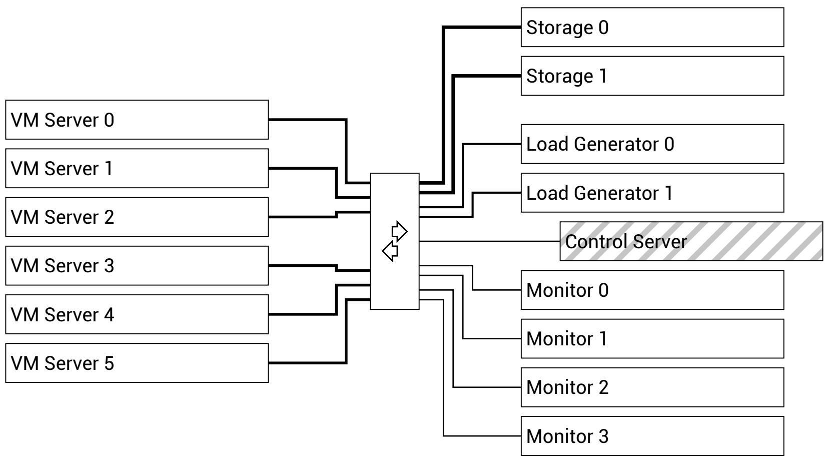


Figure A.1: IaaS testbed infrastructure

and three 1 Gbit network interfaces in a 802.3ad LACP bond. The second storage server has an Intel Xeon E5620 CPU, 16 GB memory and 3 Gbit network interfaces in an LACP bond. Disks are set up in a RAID 10 configuration. Preliminary experiments were conducted to find the optimal RAID controller configuration. Both, the network and storage infrastructure had sufficient capacity such that they did not result in bottlenecks.

A Java Glassfish[1] enterprise application container with the SPECjEnterprise2010 application and a MySQL database server[2] are installed on each VM. SPECjEnterprise2010 was chosen because it is widely used in industry to benchmark enterprise application servers. It is designed to generate a utilization on the underlying hardware and software that is very similar to the one experienced in real-world business applications.

Two additional servers are used as workload generators. Each one is equipped with an Intel Core 2 Quad Q9400 CPU with 12 GB main memory and two 1 Gbit network interfaces in an LACP bond. A modified version of the Rain[3] workload generator is used to simulate workload on the SPECjEnterprise2010 applications within the VMs.

[1]http://glassfish.java.net/
[2]http://www.mysql.com
[3]https://github.com/yungsters/rain-workload-toolkit

Four servers with heterogeneous hardware are running the Sonar monitoring system, Hadoop, and HBase. All are equipped with one 1 Gbit network interface, between 4 GB and 16 GB of RAM, an Intel Core Quad CPU Q9550 2.66 GHz, Intel Core 2 CPU 6700 2.66 GHz, or an AMD Phenom x4 905e CPU, one system disk and dedicated storage disks. The Hadoop cluster reports a total storage capacity of 7 TB.

A.2 Software infrastructure

The Rain workload generator puts the VMs in the testbed under load. The version used is an enhanced version of the Rain workload generator proposed by [7]. It was extended to cover IaaS scenarios where VMs get allocated and deallocated according to a predefined schedule as shown in Figure D.2. Chapter D provides a detailed description of the Cloudburst workload generator we developed to replace Rain in future experiments. At this point we do not explain Rain in detail as Cloudburst features identical functionality and a very similar software architecture.

The Sonar monitoring system measures resource utilizations of servers and VMs in 3 s intervals. Also, all Rain workload generators report 3 s averages of application request response times to Sonar. Chapter B describes the architecture and requirements that lead to the development of Sonar in more detail. An analysis step following each experiment reads all relevant data from Sonar. It allows a complete reproduction of an experiment and calculates a set of core metrics.

A software framework was used to implement and test different allocation controllers as described in Parts I and II. In a first step, simulations are conducted for a controller. Then, promising configurations are transferred to the testbed infrastructure and evaluated by experiments.

Through the use of an abstract interface layer, controller implementations are identical for experiments and simulations. Interfaces are either implemented to control a physical or simulated infrastructure as shown in Figure A.2. In

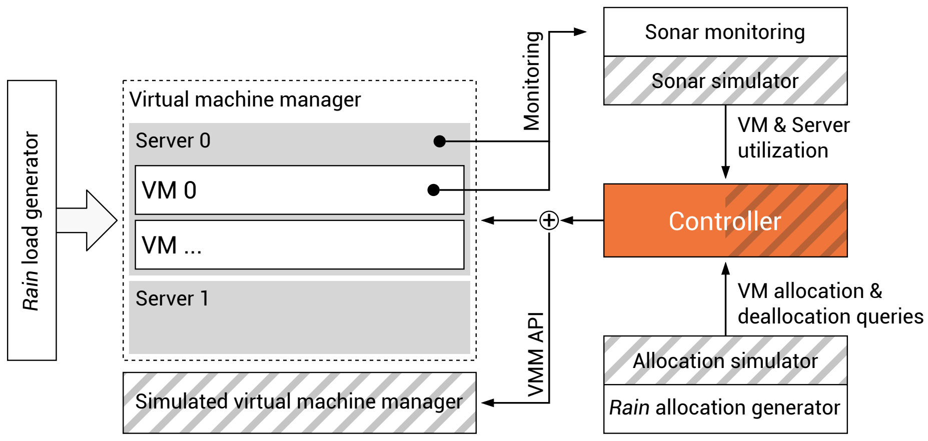


Figure A.2: IaaS framework that supports simulations and experiments

both cases, the same software framework is used. This allows an evaluation of controllers by simulation and experiments without changes.

In experimentation mode the controller receives monitoring data on all VM and server resource utilizations through Sonar. Dynamic controllers can leverage this data to trigger VM migrations on the testbed. In addition, a controller receives VM allocation and deallocation requests from the Rain allocation generator. Requests are translated into appropriate VM start and stop requests on the testbed infrastructure. For example, placement controllers as described in Part II determine where to start a new VM.

For simulation mode, all testbed services like Sonar and Rain as well as the testbed infrastructure itself are substituted by simulated components. A simulated workload generator component replaces Sonar. A time series is attached to each VM and describes its utilization over a certain time, e.g. 12 h.

Application requests (e.g. HTTP or CORBA queries) are not simulated by the framework. Simulations are based on resource utilizations of servers and VMs according to predefined time series data. Time is simulated in discrete 3 s time steps. For each step the VM utilization is determined using the time series data. Server utilizations are calculated by summing up VM utilizations. A server is over-subscribed if its CPU utilization exceeds 100 %.

Simulations leverage an VM allocation driver substituting Rain. It directly issues method calls to the IaaS service layer by using the same VM allocation schedules as used by Rain.

VM migrations are simulated by waiting a negative-exponentially distributed time with $\mu = 3.29, \sigma = 0.27$. In addition an increased CPU utilization of 8 % and 13 % is simulated on the migration source and target servers during a migration. Parameters are based on findings of Part I.

Simulated service quality (SQ) is calculated by dividing the number of simulated time intervals where a server was over-subscribed and dividing it by the total number of simulated time intervals. Contrary, *experimental service quality (SQ)* is calculated dividing the number of failed or late HTTP/CORBA requests by the total number of triggered HTTP/CORBA requests.

A.3 Time series storage

Times is a network service that is dedicated for storing time series data. For each time series it stores a list of measurements, a frequency, and a UNIX timestamp for the first measurement. A Thrift service provides functionality to search, create, and download time series data. In addition, Thrift is used to serialize a time series into a highly efficiency binary format that is stored on disk. If a time series gets downloaded, its binary file is load and written onto the network stream without deserializing it first. This implementation guarantees a high storage and network transfer efficiency.

Many components and analysis scripts directly access Times. For example, the Rain workload generators download workload profiles from Times that describe the number of users to simulate over time. All optimization-based controllers in Chapter 2 leverage Times to download workload profiles that are used to calculate an allocation of VMs to servers. As another example, analysis scripts written in R download the time series that are described in Chapter E to calculate descriptive statistics.

Appendix B

Sonar testbed monitoring

To compare the efficiency of resource allocation controllers in our testbed infrastructure we needed a tool that closely monitors it. Different metrics like CPU and memory utilization of VMs and servers, HTTP request throughput of the Rain load generator, or HTTP request response times needed to be captured. The monitoring solution had to support dynamic environments, were servers and VMs are allocated and deallocated frequently.

In contrast to data center monitoring tools, our goal was to monitor the infrastructure in a high frequency. For example, recording CPU, network, disk, and memory utilization of all systems in 3 s intervals. The system had to store this data over an extended period of approximately 3 years. All monitoring data needed to be available centrally for analytical scripts that work on application logs and time series data.

Due to the lack of appropriate monitoring solutions we designed our own system called Sonar. Its architecture is founded on the design and ideas of existing solutions in the field of data center monitoring. It differs from state monitoring systems as it is not designed to trigger alarms or to drive infrastructure health dashboards. Instead, Sonar stores RAW log traces and time series data. It is designed for simplicity of use, horizontal scalability to handle high data throughput, and high monitoring frequency below 1 min intervals without downsampling older data.

Texts in this chapter are based on a previous publication [96].

B.1 Requirements

During preliminary experiments we gathered a number of requirements regarding a monitoring solution:

Efficient — Monitoring has to be efficient to support a high monitoring frequency. System parameters have to be sampled in a frequency between 3 s and 1 min. Up to 20 metrics are monitored per server. A high throughput of monitoring data is expected, e.g. 90 measurements per second for 18 servers. In addition, application logs have to be captured at a reasonable log level. Monitoring runs continuously over a long time period and is never paused. Hence, Sonar has to be built for efficiency from ground up in regards to storage formats as well as telemetry.

Small resource footprint — The monitoring itself must not distort utilization measurements by putting the server under a high CPU or memory load. The average load of an idling server with monitoring enabled must not surpass a mean CPU utilization of 2 %. Sonar has to be reconfigurable during operation. This means enabling, disabling, installing, or reinstalling sensors automatically during operation.

Log messages and time series data — Storing of log messages and unstructured data has to be supported. All components of an experiment have to dump their configuration into Sonar for a subsequent analysis. It is necessary that such information is stored in a human readable text format. JSON serialized objects proved to be a viable option. By this way, data can be indexed and searched by a full text search engine as well.

Service orientation — Searching and fetching monitored data has to be equally easy as loading a CSV file. In Sonar all data is aligned by the real-world timeline. Each entity like a metric reading or a log message is stored with the Unix timestamp of its creation. A hostname and a sensor name identify its origin. Consequently three values are required to query data: timeframe, hostname, and sensor name.

Data archive — Typical meta data about an experiment includes its name, date, and a description. Additional attributes depend on the concrete experiments and the requirements. Sonar focuses on providing a stable data store for experimental monitoring data only. In addition, it is easily possible to use a meta data store on top of Sonar which perfectly fits the specific requirements of ones research. This can be anything from a relational database to a spreadsheet.

B.2 Related work

A lot of data center monitoring solutions exist today but do not fit our special requirements. Their design goal is to detect errors for system administrators. Examples are Nagios[1], Collectd[2], and Munin[3]. They provide rudimentary monitoring of CPU, memory, and disk utilization to indicate if these resources will become a bottleneck or if they will fail. Readings are only conducted on 5 min to 60 min intervals. A functionality to reconfigure or install sensors on the fly with automatic (re-)deployment during operation is not supported. Usually, data gets downsampled and ultimately removed to reduce required storage capacity.

Ganglia[4] and Astrolobe [91] are designed to handle a high degree of monitoring detail but cannot store extensive amounts of data either. Each node is storing all of the monitored data which gets cloned using a peer-to-peer approach. This ensures a high data redundancy and safety in case of catastrophic failures but puts additional load on the servers which might distort utilization measurements. Monitoring data gets removed after some time in order to fit the data on a single node. Virtualized cloud infrastructures in our research

[1]http://www.nagios.org
[2]http://collectd.org
[3]http://munin-monitoring.org
[4]http://ganglia.sourceforge.net

are subject to constant change which would require the peer-to-peer system to clone servers frequently.

Chukwa[5] is capable of storing time series data as well as application logs. Its primary design goal is resilience against failures. All data is stored locally on the monitored system as well as in a central Hadoop distributed file system (HDFS). Search and browse functionality requires additional mechanisms not provided by HDFS or Chukawa. Analysis scripts have to be designed as MapReduce jobs which suits analysis of big data but makes analysis unnecessary difficult if only a small fraction of the data is analyzed.

INCA[6] pursues similar goals as for example Nagios but in the context of a Grid environment. It is not built to handle dynamic environments. Sensors and the structure of monitored data need to be configured in advance which complicates its use in dynamic scenarios. Similar to many other systems, INCA stores data in a relational database which limits storage scalability.

Other systems exist for storing application logs but do not support resource utilization monitoring. Apache Flume[7] provides a configurable plugin and data sink concept for routing data within the infrastructure. Scribe[8] is designed to stream application logs from a large number of servers. It does not support querying as the data is written to HDFS without any indexing. Otus and its successor vOtus [69, 68] are application log aggregators which are based on a MySQL database. Graylog2[9] uses the JSON format to transfer log data from the servers to the monitoring system which stores them in a MySQL database. Again, storage capacity is limited by the relational database.

OpenTSDB [80] is the system which closest fulfills our requirement of a research monitoring solution. It focuses on storing large amounts of measurements in an HBase database. Data is never down-sampled or deleted and can be queried efficiently by an HBase row scan operation. OpenTSDB is designed

[5]http://wiki.apache.org/hadoop/Chukwa
[6]http://inca.sdsc.edu
[7]https://cwiki.apache.org/FLUME
[8]https://github.com/facebook/scribe
[9]http://www.graylog2.org

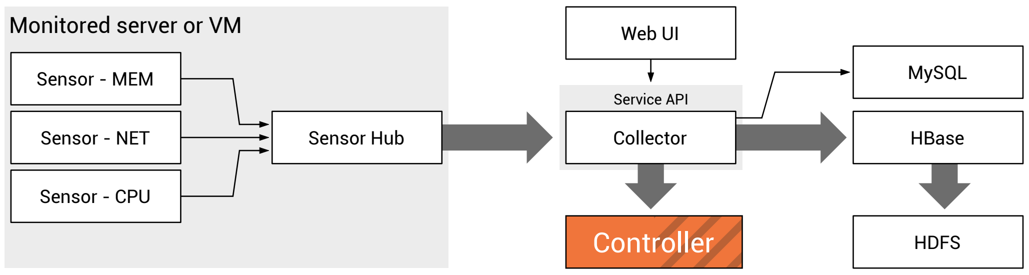


Figure B.1: Overview of the Sonar components and their dependencies

for metric data elements only, application logs are not supported. Dynamic reconfigurability of sensors is not supported, they have to be deployed and started manually on each node. It is necessary to register metrics before their usage which additionally complicates the setup process of an experiment. Data can be passed to OpenTSDB by an HTTP or Telnet based asynchronously handled RPC call. However, HTTP requires a high processing overead on server and client side and both, HTTP and Telnet are not bandwith efficient compared to binary protocols like Google's ProtoBuf[10] or Apache Thrift [82].

B.3 Sonar architecture

In the following we will describe the architecture of Sonar which leans itself to the implementation of OpenTSDB. Figure B.1 depicts its overall architecture.

All data flows from the left to the right starting with the Sensor components. A Sensor measures data and passes it to the SensorHub which transforms it and forwards it to the Collector. The Collector is designed to scale horizontally, which allows the system to scale with the number of monitored servers. It receives data from all SensorHubs and writes it into the HBase database which in turn stores everything in HDFS. MySQL is used to hold a configuration management database (CMDB) as well as the binaries of all Sensors.

In contrast to OpenTSDB, Sonar solely uses Apache Thrift for telemetry. Each

[10]http://code.google.com/p/protobuf

metric reading and log message is passed to the Collector using a Thrift RPC call. Thrift was chosen because of its high serialization and deserialization speed, its efficient storage format and its support for a wide range of programming languages [82].

B.3.1 Sensor and SensorHub components

A Sensor is a small program whose job is to read system metrics such as memory or CPU utilization, e.g. by using PAPI-V [23]. Each Sensor is executed in a separate process managed by the SensorHub. The communication between a Sensor and the SensorHub happens through the standard process input and output streams. A Sensor encapsulates each reading in a single line of comma separated text and flushes it to the standard output stream. The SensorHub listens and reads the data from multiple Sensor standard output streams using the select and poll operations of the Linux Kernel in order to keep the resource footprint low. All received data is serialized in a Thrift object. For an increased TCP/IP package utilization, a single Thrift object aggregates multiple measurements that are sent to the Collector.

In the bootstrapping sequence a SensorHub determines its hostname. By using the configuration API of the Collector it checks whether the hostname is already registered in the CMDB and does so if not. Subsequently it fetches a list of Sensors assigned to the hostname. Sensor programs are packed in a ZIP file and stored in the CMDB as well. The SensorHub downloads all relevant Sensor packages, extracts and executes them with their configuration settings from the CMDB. The standard output stream of the new process is registered internally.

In order to keep the Sensor packages up to date, the Sensor list is reloaded each 30 s. A Sensor binary is reloaded and restarted if the MD5 hash of the local Sensor package is not equal to the one stored in the CMDB.

Sensors can also be used to monitor log files and transfer new lines to the Collector. However, application logging is typically integrated in the application itself. For Java applications like JBoss, Glassfish, and Apache Tomcat a custom Log4J appender is provided which sends the application logs to the Collector. Most of our scripts are written in Python. These scripts use Python logging to dump their configuration settings and their results into Sonar by writing JSON serialized objects as log messages.

One important aspect while designing the SensorHub and the Sensors was their own resource footprint. On an idling entry level server with a single quad-core CPU, 16 GB of memory and 1 Gbit Ethernet connection, the 95th percentile of the CPU utilization remains below 1 %. The memory consumption is difficult to determine due to shared libraries. Linux reports a resident set size of about 15 MB for the SensorHub process plus 7 MB for each Sensor. For 5 Sensors this is approximately 50 MB of memory consumption, about 0.3 % of the total 16 GB available memory. The network traffic remained below 50 kBps.

B.3.2 Collector component

The Collector is a central component in Sonar. It provides a number of network services for monitoring as well as for configuration. It receives all logs and metric measurements from SensorHubs and applications.

A configuration service is used by a WebUI which provides a user interface to configure and monitor the whole infrastructure. It can be used to assign Sensors to hostnames with a specific configuration, to query and plot time series data, or to display log data. All configuration changes affect the operation of the infrastructure almost instantaneous. For example, if a Sensor is applied to a hostname, the corresponding SensorHub will automatically download and (re-)start it.

The Collector provides a relay functionality which can be used by any software component to subscribe for utilization measurements of a specific hostname and metric. Experiments in our testbed infrastructure often comprise programs which take actions based on resource utilization measurements of servers and VMs. For example, a dynamic controller migrates VMs between servers in dependence to their utilization. Usually, such components discover the infrastructure state on their own. However, this is already accomplished by Sonar which simplifies the development of such components and unifies data acquisition.

The Collector is designed to achieve a high throughput for receiving and storing metric readings and log data. It is implemented in Java and accepts Thrift packages which are stored in a processing queue first. The queue is processed by a worker thread that writes the data into HBase.

In contrast to OpenTSDB, the Collector is still based on a synchronous network model to receive data from SensorHubs and applications. A dedicated thread is used for each incoming TCP/IP connection. An synchronous model was chosen for two reasons. First, Mutsuzaki [58] showed that asynchronous processing is not optimal to achieve high message throughput using Thrift RPC. Second, each SensorHub creates a single TCP/IP connection independent to the number of Sensors. Asynchronous processing is of great benefit if the number of TCP/IP connections is extremely high as state by the C10K problem [44]. However, the number of SensorHubs per Collector is most likely below 1000 which can be handled by a synchronous approach efficiently.

To achieve high HBase throughput OpenTSDB uses a proprietary HBase client called asynchbase[11]. This client is not officially supported and newer HBase versions might not work with it. Sonar uses the official client library which is likely to be more reliable and still provides enough performance to monitor small testbeds. Throughput bottlenecks can be mitigated by scaling the Collector horizontally.

The underlying HBase table structure for time series data closely resemble the ones used by OpenTSDB but does not support labeling of measurements. The schema consists of the *tsdb* and *uidmap* table which are described in Figure B.2 and B.3. For completeness, the schema is described here, as no official documentation or reference exists except various presentation slides [81, 80].

The *tsdb* table's row-key consists of three fields: The first two elements are the hostname where the reading occurred followed by the metric name, e.g.

[11]https://github.com/OpenTSDB/asynchbase

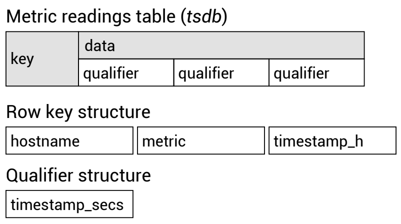


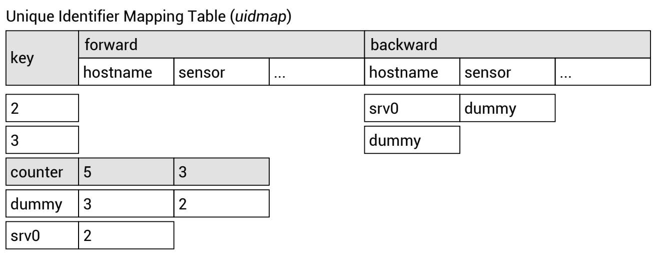
Figure B.2: HBase table schema for storing time series data

CPU or memory. A row holds all measurements which occurred during the same hour. The hour is stored in the third key-field as a Unix timestamp. All measurements are stored in separate cells in the column-family *data*. The qualifier of a cell is equal to the second(s) at which the reading occurred within the hour addressed by the row-key.

HBase sorts the data based on the row-key and uses a column oriented storage schema as described in the Google BigTable paper [17]. All values for the same host and metric are stored in one coherent block on disk. All measurements are stored in a coherent block as their timestamp grows monotonous. Sonar queries address values within a time frame for a given hostname and sensor. They directly translate to a linear scan, the most efficient operation of HBase.

Each HBase cell holds only a single numeric value of double precision. This is space inefficient as described by the OpenTSDB team [80]. The complete row-key is stored redundantly for each cell of a row. If a cell contains 4 Byte of data, the ratio of the row-key size and cell content size is imbalanced to the disadvantage of the cell content. Sonar runs a row compaction regularly as does OpenTSDB. It aggregates the data of all cells in a row in a single cell using a Thrift serialized object. The cells are removed after the compaction which eventually frees up the memory consumed by the row-keys and cells.

To save memory, the hostname and the metric name are not stored as strings but as integers in the row-key. The *uidmap* table maps strings to integers for this purpose (see Figure B.3). If a metric or hostname occurs which is not mapped, a new mapping is created on demand. The *uidmap* table uses column-families to differentiate between forward and backward mappings and


Unique Identifier Mapping Table (*uidmap*)

key	forward			backward		
	hostname	sensor	...	hostname	sensor	...
2				srv0	dummy	
3				dummy		
counter	5	3				
dummy	3	2				
srv0	2					

Figure B.3: Hbase table schema for the *uidmap* lookup table

columns for mapping different types, e.g. hostnames and metric names. Collissions between mappings with the same name but different types are avoided this way.

A forward lookup for a given string and mapping type works as follows. Lookup the row with the given string as row-key. Get the column family for the forward mappings. Finally, get the cell where the column equals the mapping type. The cell value then contains the searched integer value.

Reverse mapping works similarly. First lookup the row with row-key beeing equal to the integer value. Get the column family for backward mappings. Get the cell where the column name equals the mapping type. The cell contains the searched string value.

The identifiers are maintained in a special row with row-key *counter*. It holds a counter for each mapping-type in the forward lookup column family. Each time a new mapping is created the appropriate counter is incremented in an HBase transaction to generate a new identifier. The name *counter* is a special name which cannot be used for any mappings to rule out conflicts.

In addition to storing utilization measurements Sonar also has the capability of storing application logs. For application logs we propose a modified storage schema (see Figure B.4).

The row key is slightly modified and stores the complete Unix timestamp, not just the hour. Each log message is stored in the column-family and column *data*. The inter-message arrival time is below 1 s. Therefore, multiple log messages have to be stored in the same row. To prevent new logs from overwriting

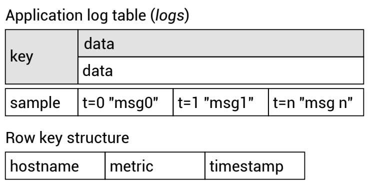


Figure B.4: HBase table schema for application logs

old log messages, the HBase versioning feature is used. Each time a cell is modified, a new version of the cell is created without deleting or overwriting existing data.

Internally HBase stores the row-key for each cell version which can be storage inefficient. The ratio of the content of a cell and the size of the row-key has to be balanced to achieve a high store efficiency. For the table *logs* a row-key consumes 12 B. We analyzed over one million log messages from Sonar. The mean message length was 229.32 characters with a standard deviation of 740.92 characters. The 10th percentile was 27, the 50th 35, and the 90th 841 characters.

These results indicate that 50 % of the log messages consume more than 280 B. The row-key is 12 B long, which is below 4.3 % of this message size. This imbalance might be mitigated by introducing a compaction process as used for the *tsdb* table. Compaction itself requires additional resources to run and slows down querying on compacted cells. For the *logs* table, the problem only affects below 5 % of all cells where the message length is below 25 characters. Therefore we decided to trade storage capacity for CPU resources and do not run a compaction process on the *logs* table.

The table schemes not only have to be designed for fast reads, but they also have to enable a scalable write performance. Writes scale with the number of HBase region servers if they are evenly distributed across the row-key space of each table. Then, write operations are distributed across the regions of a table and the region servers. In case of the metric table this is the case as described by George Lars [31]. The key space of the log tables is built in a very similar way as for the metric tables and therefore scales in the same way.

B.3.3 Structure of analysis scripts

Analysis scripts can download data from the Sonar system using the query service of the Collector. Network services are published by Apache Thrift so that analysis scripts are not forced into a specific programming language. The underlying HBase schema provides a high scan performance to guarantee a fast download of time series as well as log data.

In our case, analysis scripts are written in Python and use a custom layer on top of the Thrift bindings. Listing B.1 describes the steps needed to load a time series. Analysis scripts written in R leverage a Python bridge to query data from Sonar.

```
1 from sonarlytics import fetch_ts, timestamp
2 frame = (timestmp('16/09/2012␣20:20:41'),
3  timestmp('16/09/2012␣20:20:41'))
4
5 # Query time series
6 data, time = fetch_ts('server0', 'CPU', frame)
```

Listing B.1: Sample of a Python analysis script which queries data from Sonar

Two functions of the *sonarlytics* package are imported in the beginning. By doing so, the *sonarlytics* package automatically connects with a Collector. In lines 2 and 3 the time frame of the experiment is specified. In the example the formatted date strings need to be converted to timestamps first. Line 6 represents the load command which fetches a time series from Sonar and returns a tuple of two Numpy[12] arrays. The first array contains the actual data and the second one the timestamps for each data point in the first array. The function is parameterized with the query information: hostname, sensor name and timeframe. Now, an arbitrary analysis job can be executed based on the Numpy arrays which are the foundation for almost all scientific Python libraries. Fetching application logs works the same way as fetching time series data.

[12]http://www.numpy.org

This approach has several advantages while at the same time being as simple as loading data from a CSV file. First, all the data is available to all people involved in the project. This becomes more important as the data volume grows. In our case, the HDFS held more than 500 GB after 3 years of developing and using Sonar on a small testbed with about 30 servers and VMs. Such an amount of data cannot be emailed or sent over the network easily. Due to version conflicts it is not practicable to copy all the data or data fragments to other computers where it is used for analysis processing. Using Sonar, the data is available by means of a service to all devices with a network connection, even while the experiment is running and instantaneously after the experiment ended.

It is guaranteed that the data in Sonar is the RAW data. It has not been processed or altered in any way, so that all analysis is working on the same data.

Analysis scripts work very well if the data volume fetched by the script easily fits into the main memory. If a large amount of data has to be processed, it is better to exploit the MapReduce functionality provided by HBase and Hadoop. This allows to process the data in parallel on all HBase servers without sending all the data over the network. In this case, however analysis scripts get more complicated as the user needs detailed knowledge of the Sonar table schemes, data locality and the implementation of MapReduce programs.

Appendix C

Relay infrastructure control

We did not find appropriate tools to (re-)configure our testbed infrastructure in order to fulfill the prerequisites of an experiment. The Relay system was designed to establish such prerequisites like starting system services, deleting and loading databases, deploying applications, and synchronizing the Rain workload generators before an experiment is triggered as well es during an experiment where new VMs get allocated by the IaaS service.

Relay exploits parallelism to speedup the testbed setup process, so that multiple servers and VMs are set-up in parallel. Due to the high degree of parallelism with multiple servers and VMs such a process gets complicated to implement. It needs to support research environments with continuously changing requirements where flexibility and prototyping are important. For this purpose we implemented Relay. It leverages behavior trees as an alternative to finite state machines (FSMs) in conjunction with asynchronous network programming techniques.

This work is based on an interdisciplinary project done with D. Srivastav.

C.1 Architecture

Conducting an experiment usually requires the testbed infrastructure to be in a certain predefined state for each experiment. Usually some kind of automated setup process is developed to achieve things like migrating VMs, starting services, or loading database dumps.

In Unix environments SSH and Bash scripting can be considered as the standard tools to achieve this. In our first attempt we faced some drawbacks using this tools in the context of a research environment. SSH connections need authentication which has to be configured correctly for all servers and VMs. For our testbed, security was of no concern, however. Writing Bash scripts and especially changing them in case of changing demands proved to be an error prone process. Our best approach was to follow the keep it simple and small concept (KISS) and split the code into fine granular modules to keep it maintainable and stable. Still, a central Bash script was needed to trigger all the smaller scripts on a number of difference servers in parallel. The central script often had to wait for a specific service to start and generate some type of a log message. This introduced the need for concurrency and synchronization which are hard to implement and debug in Bash scripts.

We introduced a service called Relay to bypass SSH connections and Bash scripting on a high level. Each server runs a Relay service that publishes a Thrift network service. It provides only a few functions to transfer a ZIP package to the service. The package must contain a main file that can be a Python script, Bash script, or any executable binary. Relay executes the package by extracting it and launching this main file.

The package execution function blocks until the package finished its execution. An alternative function launches the package and immediately returns its process identifier (PID) without waiting for the process to finish. It can be used in conjunction with other service functions to check the process state or to terminate it. Such a functionality is required to launch server processes which will run for a longer period of time or to run a database dump script.

Sometimes it is required to wait for some text to occur in the standard output stream of a process. If this process terminates and the text was not found in its output stream the package is re-executed. Restarts are limited by a predefined number, however. If all restarts are used up an error is returned to the caller. For example, this mechanism is used to wait for a domain to become available after starting the Glassfish server.

A similar function exists which starts the package only once and then scans its standard output stream for a message. The API call returns with the process PID as soon as the message is found or until a timeout occurred without terminating the package execution. Alternatively, the function executes the package and opens a file handle on a log file instead of the standard output stream.

Relay is implemented as an asynchronous Thrift service to keep the memory and CPU footprint low. The asynchronous implementation style eliminates all multithreaded and therefore deadlock related problems of synchronous server and client implementations. The concepts are explained in the following.

The reconfiguration process of an infrastructure is described in a single Python script by triggering asynchronous function calls. Each function transfers and launches a ZIP package on a Relay service instance. As the asynchronous calls are non-blocking, all ZIP packages are executed in parallel. Deferred lists are used to synchronize calls by waiting for multiple asynchronous functions to return.

```
def function finished():
    pass

def function deploy(connections):
 dlist = []

 # Launch Glassfish servers
 for server in hosts_list:
  dlist.append(relay.launch(connections, server,
   'glassfish_start', wait=False))
  dlist.append(relay.poll_for_message(connections,
   server, 'glassfish_wait', 'domain␣1␣running'))

```

```
14 # Load database dumps
15 for server in database_list:
16  dlist.append(relay.launch(connections, server,
17   'spec_dbload'))
18
19 # Wait for all operations to finish
20 defer.DeferredList(dlist).addCallback(finished,
21  client_list)
22
23 def function connect():
24     dlist = connect_all(hosts_list)
25     defer.DeferredList(dlist).addCallback(deploy)
```

Listing C.1: Sample implementation of a testbed setup process

Listing C.1 describes a simplified version of such a setup process script. The connect function establishes a connection with an arbitrary number of servers running a Relay service. In line 25 a deferred list is used to wait for all connections to be established and calls the *deploy* callback handle afterwards. The *glassfish_start* package is used to start an instance of the Glassfish service on all Relay connections in line 9. Setting the wait flag to *false* tells Relay not to wait for the package execution to finish and the function call returns immediately with the process PID. The *glassfish_wait* package is executed on all Relay connections as well. It waits until the primary Glassfish server domain is available and then returns the call. A domain is running if the Glassfish command line interface tool prints the string "domain 1 running". Finally, in line 16 the *spec_dbload* package is executed on all connections. It loads a database dump for the SPECjEnterprise2010 application and then returns.

Due to the asynchronous programming style all remote functions calls immediately return a deferred object. This object is used to register a callback handle that gets called as soon as the asynchronous function call returns a proper value. In the example, a deferred list is used in line 20 to group the deferred objects from multiple remote function calls. The list calls its own callback handle *finished* if all its asynchronous calls returned.

In summary, the setup process waits until Glassfish is up and running and the database dump is loaded on all connected servers. After that, new remote function calls can be triggered within the *finished* function.

We found asynchronous programming to simplify the programmatic description of such setup workflows that incorporate many servers and Relay connections. Creating small independent packages like starting a Glassfish service enabled us to describe a process by chaining the execution of existing reusable packages.

The program structure resembles a FSM which does not contain any code to handle concurrency. The workflow is described by chaining callback handlers of deferred objects or lists together (see Listing C.1). For larger programs this proved to be difficult to maintain as the definition of the workflow is spread across the whole program and leads to an uncontrolled growth of new program functions.

C.2 Behavior trees

Behavior trees [48] let us describe the complete workflow in a single place. A behavior tree consists of action and control nodes. Each node returns a binary status after its execution. Sequence control nodes execute all their child nodes sequentially from the left to the right. The execution is stopped if one node returns a *false* execution status. Then the sequence node returns *false* too. Probe control nodes execute their child nodes sequentially from left to right until a node returns *true*. The remaining child nodes are not executed and the probe node returns *true* too. A probe node returns *false* if none of the child nodes executed successfully (returned *true*). A concurrent control node executes all child nodes concurrently and waits for all childs to return. A logical AND operation on all child execution statuses gives the return value of the concurrent node. Action nodes are a wrapper around the existing packages described before. They execute a concrete operation on one or multiple Relay service instances.

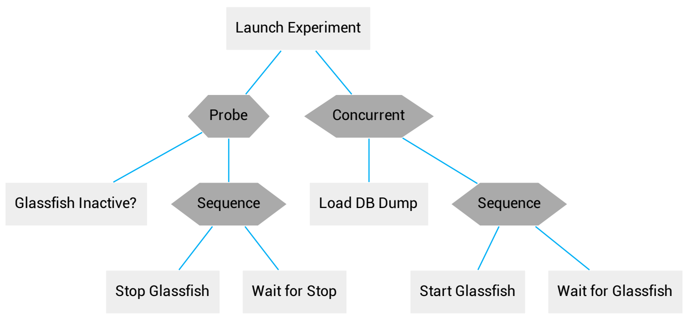


Figure C.1: Infrastructure setup workflow description as a behavior tree

Figure C.1 shows an example of a behavior tree. Its execution starts at the root node of the type sequence. It calls the leftmost probe node which in turn calls the first action node. The action node checks whether all Glassfish servers are stopped and returns *true* if so. Otherwise, it causes the probe node to call its next child node, another sequence node. It triggers two action nodes which stop all Glassfish servers and wait for them to terminate.

The second subtree of the root node is based on a concurrent node. It executes an action node that loads a database dump as well as a sequence node in parallel. The sequence node has two action nodes attached which start the Glassfish server and then wait for it to become available. The parallel node returns *true* if the database dump is loaded and all Glassfish servers are running.

Our implementation of behavior trees is based on asynchronous programming as well. A child node that gets called does not block the call but returns a deferred object instead. The calling node might register itself as a handle in the deferred object so that it gets called if the child node finishes its work. This approach has two advantages. First, action nodes trigger package executions and are already implemented asynchronously which very well integrates with the asynchronous behavior tree implementation. Second, it is straight forward to implement concurrent nodes without threads and synchronization by using asynchronous programming.

Summarized, behavior trees completely separate the technical implementation aspects of asynchronous programming from the workflow description. Still they can be described in Python's data structures in a readable and concise way. A high degree of reusability can be achieved as subtrees can be part of multiple behavior trees. Changes in a complex behavior tree do not risk the stability of the whole tree but only a subtree. In addition, testing of complex workflows is simplified as each subtree can be tested independently.

Appendix D

Cloudburst

One challenge we faced in building and using our testbed infrastructure was, how to simulate realistic load on the applications inside VMs. A number of well-known tools such as Faban[1] and Rain[2] exist that provide some but not all functionality to generate realistic workload. First, an IaaS workload generator has to generate variable load over time on multiple target VMs. Second, it must allocate and deallocate VMs from the cloud infrastructure dynamically according to a predefined schedule. Third, it needs to keep track of several statistics and counters for later analysis such as the average request throughput and response time or the total number of triggered requests.

Another problem is to provision enough hardware to generate sufficient load on all VMs in the testbed. We found that Faban and Rain utilize lots of memory while CPU was not a bottleneck in our case. Both are based on Java and create a new thread for each simulated user. Lots of users are required for each VM to generate realistic load. Each thread introduces additional memory and CPU overhead. Using a different paradigm of parallel programming such as provided by Erlang, Scala, or Go may lead to an performance improvement with a reduced memory footprint.

[1]http://www.faban.org
[2]https://github.com/yungsters/rain-workload-toolkit

We implemented Cloudburst[3], a load generator that is designed to generate workload for IaaS cloud environments. It is developed in the Go programming language that allows simulating a huge number of users in parallel. Goroutines substitute threads and consume less memory.

Our finding is, Cloudburst uses significantly less memory while generating the same workload as Rain. Rain has a lower CPU utilization, most likely due to the high optimization degree of the Java VM. If memory is a scarce resource compared to CPU, Cloudburst and Go can simulate more users with the same hardware.

In Section D.1 we discuss related work such as Rain and Faban. Different programming languages with alternative programming paradigms for concurrency such as Erlang, Scala, and Go are discussed in Section D.2. The architecture and components of Cloudburst and the SPECjEnterprise load generator implementation are described in Section D.3. Section D.4 presents an experimental evaluation by comparing the performance of Rain and Cloudburst.

Texts in this chapter are based on a previous publication [50].

D.1 Related work

Faban is a load generator that simulates dynamic load on a single target system. Dynamic load varies the number of simulated users on a target system over time according to a given time series as shown in Figure D.1. In contrast, traditional peak performance benchmarks such as ApacheBench[4] only simulate a constant number of users.

With Faban we found it hard to generate load on multiple targets at the same time as multiple load generator instances have to be configured and started synchronously. Its software architecture renders it virtually impossible to implement more advanced features required by an IaaS load generator. An

[3]http://www.github.com/johanneskross/cloudburst
[4]http://httpd.apache.org/docs/2.2/programs/ab.html

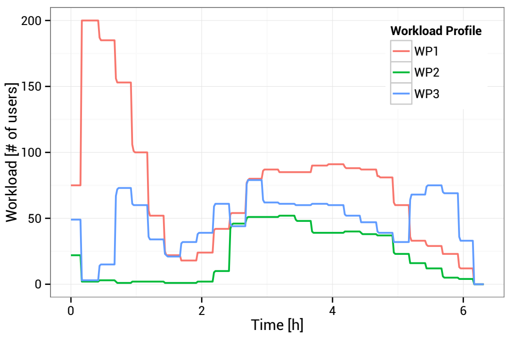


Figure D.1: Three workload profiles that describe the number of simulated users over time that are used by the load generator

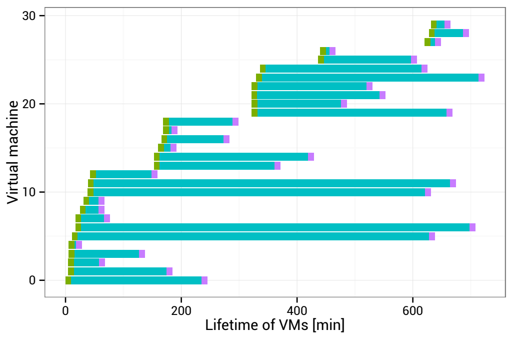


Figure D.2: VM arrival and departure schedule that is used by the IaaS load generator

opinion that is shared by Rean Griffith[5] and lead to the development of Rain [7].

Initially we extended Rain to provide all features required by an IaaS driver. We'll refer to Rain as this extended version[6]. It loads a VM schedule that describes the temporal sequence of VM allocations with their lifetimes as depicted in Figure D.2. After allocating a VM, it is initialized by installing and starting applications. Then, dynamic workload is generated towards it following a predefined workload profile as shown in Figure D.1.

Faban and Rain are based on a thread per user architecture. Both tools start a thread pool with the size of the maximum number of users that will be simulated in parallel over the benchmark. Depending on the current number of simulated users some of the threads are blocked so that the number of active threads resembles the number of active users for each target VM at any time. In Faban each thread represents a single user while Rain decouples threads and users which allows more flexibility regarding closed- and open-loop load generators. Open-loop generators allow a asynchronous processing of requests without actively waiting for a response.

[5]http://www.youtube.com/watch?v=qgZ1jtnrIEs
[6]http://www.github.com/jacksonicons/rain

Each thread exhibits a constant memory and CPU footprint for managing its state and context switches. Our goal was to simulate as many users on as many target VMs as possible with minimal server hardware. We found that Rain required a considerable amount of memory while CPU was not a limiting resource in our case. In order to minimize the memory footprint we looked into programming paradigms that are designed for highly concurrent applications.

D.2 Background

Load generation comprises a high amount of concurrency which is the most important aspect while choosing the underlying technology and software architecture.

Rain is implemented in Java and leverages threads for concurrency. “A Thread is a basic unit of program execution that can share a single address space with other Threads - that is, they can read and write the same variables and data structures” [72]. According to the Oracle Java VM documentation[7] each thread allocates between 256 kB and 512 kB to hold its stack depending on the system platform. This value can be set manually by the *-XX:ThreadStackSize=512* parameter of the Java VM.

In Rain all simulated users are independent which keeps thread synchronization at a minimum. At certain points like updating central counters and recording metrics synchronization is inevitable. Here, queues minimize wait times so that synchronization does not cause performance bottlenecks of any kind. All queue buffer sizes are monitored and size limited to rule out memory leaks.

Carl Hewitt [93] proposes the actor model as an alternative to the thread model. Each object represents an actor that comprises a mailbox and a behavior. Multiple actors run concurrently and send messages to others actors’ mailboxes. If an actor receives a message it calls its behavior which might trigger another message. All message transfers happen asynchronously so that senders and receivers are not blocked. In contrast to threads where locks are

[7]http://www.oracle.com/technetwork/java/javase/tech/vmoptions-jsp-140102.html

used as controlling instances, the actor model uses messages without shared variables.

Erlang[8] is a functional programming language that is built upon the actor model in order to provide high concurrency at a low resource footprint [93]. Actors and messages are scheduled internally by Erlang and not by the operating system. Each Erlang process runs on a single CPU core independent to the number of actors. Multiple Erlang processes are required to fully utilize a many CPU and core architecture.

Scala[9] provides the actor model as a library to Java applications. In contrast to Erlang it supports Java object-orientation as well as functional programming. Programs are compiled into Java intermediate code and run on top of the Java VM. The actor model implementation is similar to Erlang with additional features such as a message reply function. Scala introduces the concept of channels that are used to transfer messages. A channel defines a set of message types that are allowed to be transferred. As Scala integrates with Java it supports two types of actors. A thread based actor is resembled by a Java thread. Event-based actors are processed together on a single thread. They are a lightweight alternative to the first one. Both actor types can be used within an application in order to benefit from each ones advantages.

Go[10] is a compiled language designed for concurrent programming, multicore processors and networked systems [64]. It leverages communicating sequential processes (CSP) as originally proposed by Tony Hoare [41] that remind at Erlang and Scala. Functions that are able to run concurrently with other functions are called goroutines. Variables are shared by sending them around in channels so that they are always owned by a single goroutine and locking is unnecessary. Channels are either buffered or unbuffered. If data is transmitted by an unbuffered one, the sender has to wait for the receiver. For buffered channels, sender and receiver are decoupled by a buffer.

[8]http://www.erlang.org/
[9]http://www.scala-lang.org/
[10]http://golang.org/

According to latest benchmarks[11], creating a new goroutine entails a memory overhead of approximately 4 kB to 5 kB. Because they share the same address space they are a lightweight alternative compared to Java threads that demand between 256 kB and 512 kB as noted above.

Other aspects have to be considered if choosing one of the concurrency models discussed so far. Erlang is made for concurrency and scalability but its functional programming style seems to limit the number of available developers. Scala is object-oriented and integrates seamlessly with Java code. As most of our load generating code and especially the SPECjEnterprise2010 benchmark are written in Java this would be beneficial. However, Scala runs on top of the Java VM and the integration of functional programming with Java's object oriented features seems to increase complexity. In addition, all load generator deployments would require a Java VM with multiple JARs of the load generator framework, load driver implementation, helper libraries such as Apache Commons, and a start script to configure the Java classpath.

Go in contrast is designed to be as simple as possible with a low footprint and installation overhead. Programs are compiled into a single executable. It supports a huge degree of parallelism by supporting similar models as Erlang and Scala. Its syntax is similar to Python and C. Because of this aspects we developed a new load generator in Go. At some point this decision is also based on personal preferences. As Go uses similar models than Erlang and Scala we argue that our experimental results are transferable to other languages. Especially as Go is a relatively young language and has not undergone as much optimization as any of the alternatives mentioned beforehand.

D.3 Architecture

In this section we describe the system architecture of the Cloudburst load generator that is similar to the one of Rain. It consists of several components as shown in Figure D.3.

[11]http://en.munknex.net/2011/12/golang-goroutines-performance.html

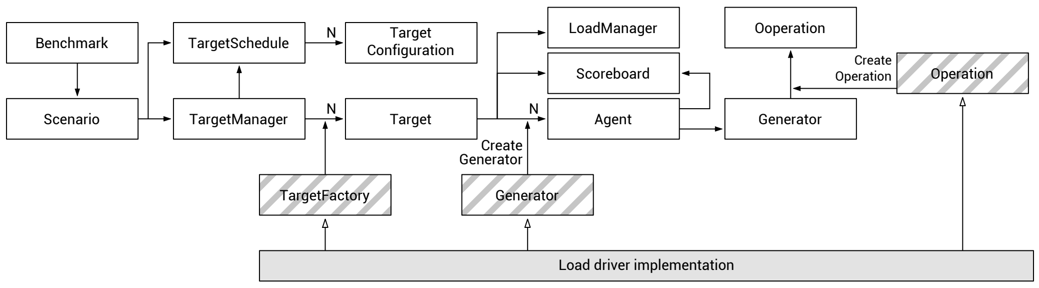


Figure D.3: The Cloudburst load generator architecture with a load driver implementation

System configuration settings are passed to the *Benchmark* that initializes a new *Scenario*. A *Scenario* coordinates the execution of a benchmark. It creates a *TargetSchedule* and a *TargetManager* instance that in turn governs multiple *Target* instances.

TargetSchedule parses the benchmark configuration and creates multiple *TargetConfiguration* instances, one for each VM that will be used during the benchmark. For each target multiple configuration settings like ramp-up, ramp-down, load duration, and the load profile are stored.

A *TargetManager* manages the lifecycle of all *Target* instances. It waits for the *TargetSchedule* until a new *Target* should be created. Timings depend on a schedule configuration as shown in Figure D.2. New instances of *Target* are created by the *TargetManager* leveraging the *TargetFactory*. The *TargetManager* waits until all *Targets* reached their end of life and then terminates itself.

Each *Target* instance is bound to one VM. It allocates and deallocates the VM from the IaaS layer and initializes it. Then, it generates load on the applications running within the VM. Load is generated by an instance of *LoadManager* that schedules the allocation and deallocation of *Agent* instances. The number of *Agent*s over time is described by the load profile as shown in Figure D.1.

An *Agent* sends queries towards a VM. The next query to run is chosen by the *Generator* which is shared amongst all *Agent*s of the same *Target*. Each time the *Generator* is called it returns a new *Operation*. An *Operation* comprises a run-method that actually triggers queries against an application. After an operation has been executed, an *Agent* waits for some time until it requests another *Operation*.

Scoreboard instances receive and aggregate operational statistics of a *Target*. Such statistics comprise waiting times and number of successful and failed operations.

A separate **load driver** component provides an implementation of how actual workload is generated on the VMs and IaaS services allocate and deallocate VMs. For that, it implements the *TargetFactory*, *Generator*, and *Operation* interfaces. It extends the abstract *Target* class to implement the interaction with the IaaS layer in order to allocate, deallocate, and initialize VMs.

D.3.1 Differences to Rain

The basic architecture is fairly similar between Rain and Cloudburst and both support exactly the same feature set. Most noticeable differences are the way *Agent*s are managed and how concurrency is realized. In Rain, a *Target* starts new *Agent* threads while the number of threads is equal to the maximum number of users that will be used in parallel during the benchmark. All threads are created while initializing the *Target*. Each *Agent* is in one of two states: active or inactive. Active *Agent*s run and execute *Operation*s while inactive ones are idle and wait for their reactivation. The *Target* adjusts the number of active and inactive *Agents* in compliance to the load profile.

In Cloudburst, a *Target* only starts the number of *Agents* that are required at a time. New *Agent*s are started if the number of users in the load profile increases while existing ones are terminated and garbage collected with a decreasing number of users in the load profile.

Since Rain is written in Java and Cloudburst in Go, they are based on two different concurrency models and use different ways to communicate between concurrent entities. In Rain, objects and variables are shared by threads. For instance, one *Scoreboard* is shared by all *Agents* of each *Target*. In order to report results from *Operation*s, the *Scoreboard* uses synchronization and locks certain variables for an exclusive access.

In Go, channels are used to share object-pointers between goroutines. Taking the same example, a *Scoreboard* holds a channel where it receives results and processes them one at a time. This channel is shared by all *Agent*s of the same *Target*. Explicit locking is not necessary in this case.

D.3.2 SPECjEnterprise2010 load driver

Cloudburst and Rain workload generators only provide the framework to simulate users. The operations that a simulated user performs on a VM are implemented by a load driver that is loaded as a submodule and integrates with the load generator interfaces.

We implemented a Go version of the SPECjEnterprise2010 load driver in order to benchmark Cloudburst against Rain. The same driver is implemented in Java that runs with Rain. The SPECjEnterprise2010 driver generates *Operation*s that create and send HTTP requests towards the SPECjEnterprise2010 business application running within a Java enterprise container in a VM. This application closely resembles the behavior of real-world business applications.

D.4 Experimental results

Cloudburst and Rain are evaluated in a testbed infrastructure. Both are installed on a server with a Fedora Linux 19 (64-bit) operating system, a Intel Core 2 Duo CPU E6750 at 2.66 GHz, 4 GB RAM and a 1 Gbit full duplex network interface. Go version 1.1.2 linux/amd64 and Java version 1.7.025 (64-bit) are used. Both simulate a varying number of users over a duration of 4 h according to a workload profile. Our goal was to measure memory efficiency in dependence to the workload profiles. Therefore, the VM allocation and deallocation feature was not used for this benchmark.

# Users	Workload	Replica	# Users	Workload	Replica
50	WP1	3	100	WP3	3
50	WP2	3	150	WP1	3
50	WP3	3	200	WP1	3
100	WP1	3	200	WP2	3
100	WP2	3	200	WP3	3

Table D.1: Summary over all experimental treatments to benchmark Rain against Cloudburst

The load is generated on 8 identical Windows 7 (32-bit) workstations in the same network. All are equipped with an Intel Core 2 Duo CPU E6750 at 2.66 GHz, 4 GB RAM and a 1 Gbit full duplex network interface. A GlassFish v3 application server with the SPECjEnterprise2010 application and a MySQL 5.5 database server were installed on each machine.

Three different workload profiles (WP1, WP2, WP3) were picked to test the load generators under different conditions. WP1 contains strong variations, WP2 is balanced and WP3 has mixed characteristics. All profiles are shown in Figure D.1. Each workload profile was tested using 50, 100, and 200 users by normalizing the profile to the user count so that it requires the desired number of users at its global maximum. For each profile and user combination we conducted three experiments with Cloudburst and Rain respectively. For WP1 another experiment with 150 users was conducted. Table D.1 shows the experimental treatments for all 60 experiments, each with a duration of 4 h.

For each execution we measured CPU and memory utilization on the load generating server in a 10 s interval using Linux's SAR utility that is based on SYSSTAT package[12]. SYSSTAT uses the */proc* directory in Linux to collect system performance data provided by the Linux Kernel. For the effective CPU utilization we report the %SYSTEM value minus the average of the first 6 measurements of $\%SYSTEM$ while the load generator was not running. Effective memory consumption is calculated by KBMEMUSED - KBBUFFERS - KBCACHED. This approximately gives the actual memory demand during operation. However, an exact calculation or measurement of memory is difficult

[12]http://sebastien.godard.pagesperso-orange.fr/

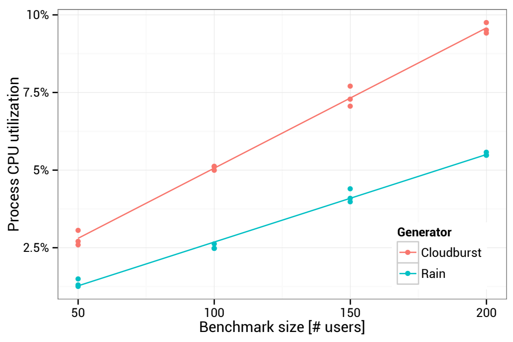


Figure D.4: Cloudburst's and Rain's 50th quantiles of CPU utilization in relation to number of users

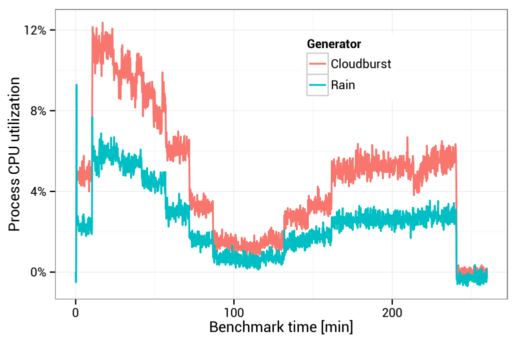


Figure D.5: Cloudburst and Rain CPU utilization over time applying WP1 normalized to a maximum of 100 users

due to various buffers and shared libraries.

D.4.1 CPU utilization

A regression of CPU utilization in relation to users is shown in Figure D.4. It indicates that the CPU utilization increases linearly with the number of users for both systems. Rain constantly consumed less CPU and its demand increased slower with additional users compared to Cloudburst. The linear regression equation for Rain was $cpu^R(u) = 0.0282 \cdot u - 0.135$ and for Cloudburst it was $cpu^{CB}(u) = 0.0452 \cdot u + 0.547$.

As an example, Figure D.5 shows the CPU usage over time for an experiment with WP1 at 100 users. For both systems, the measured CPU graphs closely resemble the workload profile that describes the number of simulated users over time. It is obvious that Cloudburst constantly consumes more CPU than Rain. At $t = 0$ it can be seen that Rain with Java consumes more CPU while starting compared to a less pronounced peak for Cloudburst with Go.

For 100 users and all three workload profiles we calculated the CPU utilization taking the 50th quantile. Figure D.6 puts them side by side. All results are summarized in Table D.2. For WP1 and WP3 the difference in CPU utilization

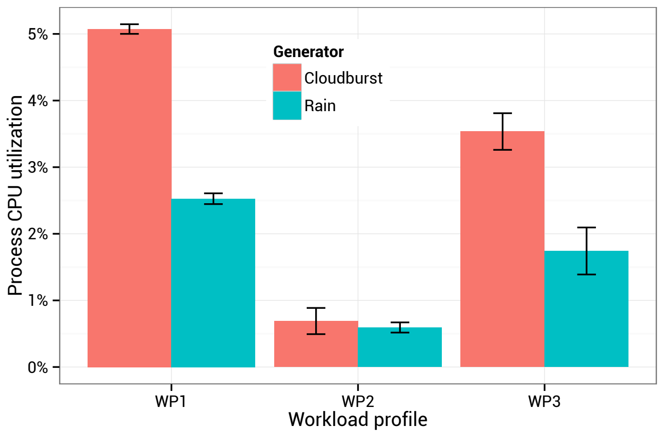


Figure D.6: Cloudburst and Rain 50th quantiles of CPU utilization over three runs using workload profiles normalized to a maximum of 100 users

was comparably large. For WP2 it was negligible, however. Most likely this is due to the workload profile characteristics as WP2 is more balanced than WP1 or WP3.

For experiments with 50 and 200 users the number of operations executed by Rain and Cloudburst differ. For 50 users Cloudburst executed 2.3 % operations more while for 200 users it executed −7.5 % operations less. We looked closer into this finding as it might indicate failures in our implementation of the SPECjEnterprise2010 driver or Cloudburst itself. An analysis of execution times reveals that Cloudburst takes 0.57 s on average for the execution of a single operation while Rain only requires 0.11 s. This difference is likely caused by inefficient implementations of the HTTP libraries and programming language. We found similar reports in the mailing list of the Go programming language. This causes an operation of Cloudburst to take longer on average compared to Rain which negatively effects the total number of executed operations.

In the following we conducted experiments with 100 users because both load generators triggered a similar amount of operations in this case.

For 100 users we applied statistical tests to see whether both CPU demands differ significantly from each other. For all experiments we took the 50th quantile of the CPU load and conducted a two-sided Student's t-test. For

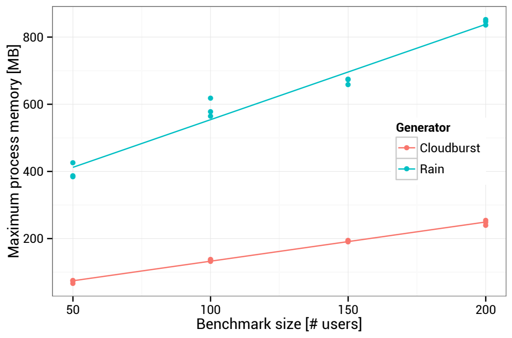


Figure D.7: Cloudburst and Rain maximum memory demand in relation to number of users

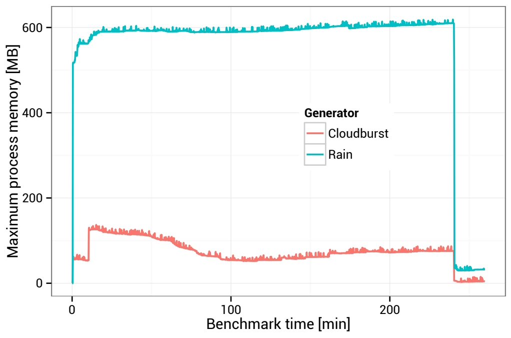


Figure D.8: Cloudburst and Rain memory demand over time applying WP1 normalized to a maximum of 100 users

WP2, it gave $t(2.59) = 0.76, p = 0.5089$ so that the CPU utilization cannot be considered as significant. However, it resulted $t(4.00) = 42.40, p = 1.882e-06$ for WP1 and $t(3.65) = 6.73, p = 0.003518$ for WP3 indicating that Cloudburst has a significantly higher CPU utilization than Rain.

One reason for this might be that Java has been optimized for CPU and memory utilization over years. Go is a relatively young language and CPU performance optimization was set on a low priority until now.

D.4.2 Memory utilization

As for the CPU, a regression of the maximum memory consumption in dependence of user count is shown in Figure D.7 for WP1 at 100 users. Comparing the mean or median memory consumption would not be entirely fair as Rain initializes all threads right away while Cloudburst initializes goroutines on-demand. Again, both systems use linearly more memory with increased users.

The linear regression equation for Rain was $mem^R(u) = 2.84 \cdot u + 270.21$ and for Cloudburst it was $mem^{CB}(u) = 1.12 \cdot u + 16.01$. So, Cloudburst required 2.5 times less memory than Rain for each additional user and 16.9 times less memory in ground state.

Memory consumption courves of a run with 100 users are illustrated in Figure D.8. It shows that Rain noticeably demands more memory than Cloudburst all the time. Differences in the garbage collection and thread creation are noticeable. Memory demand for Rain is almost constant with a slight trend upwards. In contrast, Cloudburst clearly depicts the workload profile. Most likely this is because Rain initializes all threads during its initialization and releases all of them right before termination. Cloudburst creates and deletes goroutines on-demand as the number of users changes. If a goroutine is deleted its memory is released as well which is reflected by the graph.

This is also the reason why average memory demand is a bad indicator. The maximum memory demand was used instead. It compares the memory demand of both generators during peak-load where equally many goroutines as threads are active.

Despite the different architectural design, the different memory management of Go and Java becomes noticeable comparing concrete values. Figure D.9 presents the maximum memory demand over three replicas for each workload profile. For WP2 the memory consumption for Rain was 358 MByte and the standard deviation 4 MByte. The memory consumption of Cloudburst was 41 MByte and the standard deviation was 9 MByte. A similar scenario is drawn for WP1 and WP3.

We compared the maximum memory consumption of Rain and Cloudburst for 100 users over three replications and workload profiles. Student's t-Test gives $t(2.04) = -27.68, p = 0.001159$ for WP1 which indicates that Cloudburst uses significantly less memory compared to Rain. For WP2 we got $t(2.7) = -57.06, p = 2.939e-05$ and for WP3 $t(2.7) = -26.50, p = 0.0002424$. This confirms our initial hypothesis that memory consumption of a load generator can be reduced by leveraging a concurrency model other from threads.

An extrapolation based on our CPU and memory utilization measurements indicates that Cloudburst can handle more users than Rain on our workload generating test server. Based on the linear regression models of Sections D.4.1 and D.4.2 we calculated the maximum number of users supported by our test system for 100 % CPU and 4 GB of memory utilization. Cloudburst is CPU

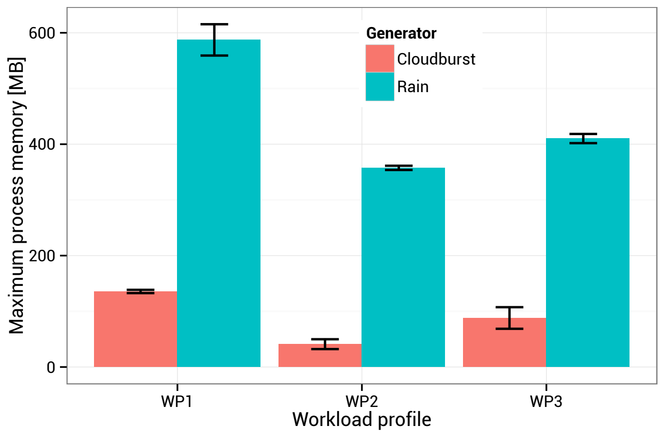


Figure D.9: Cloudburst and Rain maximum memory consumption over three replicas using workload profiles normalized to 100 users

bound and supports 2203 users (3497 for memory) while Rain is memory bound and supports only 1348 users (3553 for CPU).

D.5 Conclusions

In this work we implemented a new IaaS load generator that allows us to evaluate IaaS resource allocation controllers in experimental testbeds under realistic workloads. Such a load generator is required to simulate a varying number of users over time for multiple VMs that get allocated and deallocated on the IaaS testbed according to a predefined schedule.

We found that an extended version of the Rain load generator had a considerable memory footprint. Our objective was to cut memory consumption to generate more load with less hardware. Rain is based on Java where each simulated user is handled by a dedicated thread. Each thread introduces additional CPU and memory overhead.

We looked into alternative programming models for highly concurrent applications such as Erlang, Scala, and the Go programming language. For the following reasons we decided to implement a new load generator based on Go.

Generator	Profile	CPU [%]	Memory [MB]
		50 Users	
Cloudburst	WP1	02.96, 02.48, 02.58	073.33, 075.03, 067.00
Rain	WP1	01.41, 01.20, 01.26	426.37, 386.42, 384.41
Cloudburst	WP2	00.20, 00.31, 00.41	037.03, 038.38, 033.72
Rain	WP2	00.30, 00.26, 00.56	264.25, 282.84, 282.51
Cloudburst	WP3	01.85, 01.77, 01.31	055.07, 043.52, 043.17
Rain	WP3	00.76, 00.81, 00.75	290.93, 329.03, 315.51
		100 Users	
Cloudburst	WP1	04.84, 04.96, 04.99	136.98, 137.64, 132.23
Rain	WP1	02.38, 02.38, 02.51	618.76, 578.30, 564.74
Cloudburst	WP2	00.78, 00.41, 00.56	049.50, 041.65, 031.88
Rain	WP2	00.45, 00.55, 00.60	353.62, 357.77, 361.19
Cloudburst	WP3	03.57, 03.11, 03.37	069.18, 086.95, 107.88
Rain	WP3	01.97, 01.77, 01.25	413.68, 400.71, 416.00
		150 Users	
Cloudburst	WP1	06.86, 07.50, 07.04	190.25, 193.21, 194.84
Rain	WP1	03.88, 03.96, 04.31	674.84, 657.96, 674.06
		200 Users	
Cloudburst	WP1	09.46, 09.30, 09.15	248.71, 253.88, 238.92
Rain	WP1	05.44, 05.33, 05.36	852.41, 846.62, 835.90
Cloudburst	WP2	00.90, 00.86, 00.53	066.25, 065.96, 060.15
Rain	WP2	00.90, 00.75, 00.70	384.06, 387.83, 415.83
Cloudburst	WP3	04.92, 06.29, 05.93	108.04, 107.93, 095.00
Rain	WP3	03.32, 03.65, 03.29	459.05, 485.05, 476.51

Table D.2: Experimental outcomes on Cloudburst and Rain: CPU – median of the CPU utilization, Memory – maximum memory utilization

It has a simple and concise syntax that is similar to Python, Java, and C. Go programs are compiled into a single executable binary which does not require anything like a Java VM or additional libraries on the deployment system. Go provides lightweight parallelism by its goroutines and channels. This paradigm is similar to the ones found in Erlang or Scala. It enables us to simulate a huge number of users in parallel at a low memory footprint.

To see whether the Go load generator outperforms the Rain implementation we conducted a number of experiments in a testbed infrastructure. Our results indicate that Cloudburst consumes more CPU but less memory than Rain. According to linear extrapolations Rain is memory bound and was only able to simulate 1348 users on our test server. Switching to Cloudburst that is CPU bound allows to simulate 2203 users. In summary, Cloudburst allows us to generate more workload using the same hardware.

Appendix E

Workload data

E.1 Time series data

A similar analysis was done by B. Speitkamp and M. Bichler [86] for data set SET1. The results reported in this thesis are done one data sets SET1 and SET2 with a different interpretation. All illustrations and results for SET1 and SET2 were recreated.

Experiments are based on a set of time series data that describes the CPU utilization or application request throughput in two data centers of large European IT service providers. Speitkamp et al. [86] used a subset of this data and contributed descriptive statistics. In this section we provide similar descriptive statistics on both data sets.

The first data set (SET1) contains time series data on the CPU utilization of servers running ERP and WAD services. SET1 consists of 419 time series data files, each over a duration of 78 days in a 5 min sampling rate. Based on auto correlation function (ACF) plots we found that most of the time series show a seasonality on a daily and weekly level. Figure E.1 shows the ACF function with different lags from 288 to 26208. A lag of 288 represents 24 h. Based on the first and second subplot with lags 288 and 2016 it is obvious that this particular time series under consideration shows a strong daily seasonality.

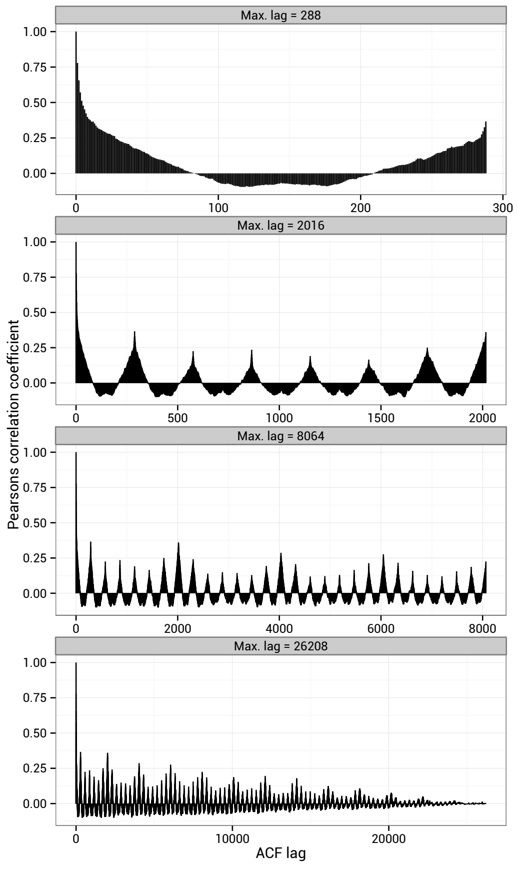


Figure E.1: ACF function with different lags over a sample utilization time series from SET1

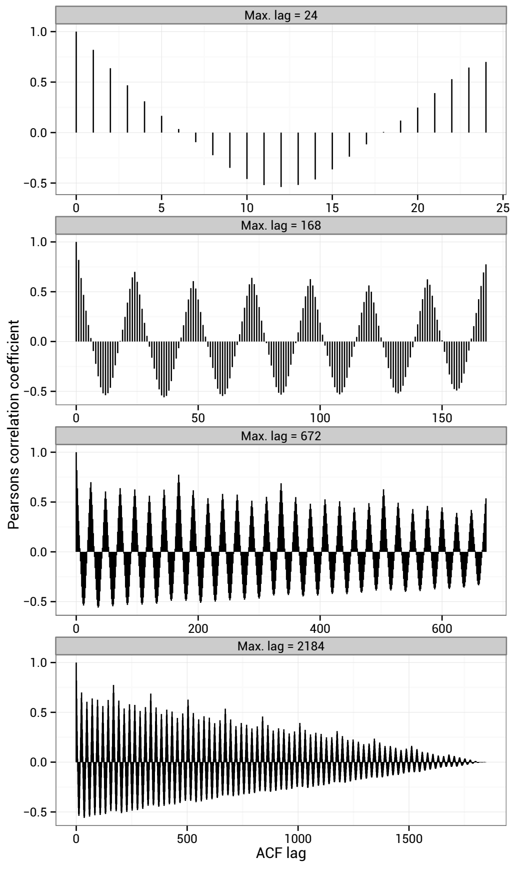


Figure E.2: ACF function with different lags over a sample utilization time series from SET2

Plots with a longer lag indicate that there is an superimposed seasonality at lag 2016 that represents a one week time frame.

A similar analysis was done for the second data set (SET2) that contains time series of web service request throughput. It consists of 32 time series with a sampling rate of 60 min over a duration of 77 days. Figure E.2 shows the ACF functions with similar results as for SET1. This time, a 24 h seasonality with a lag of 24 and a weekly seasonality at lag 168 is noticeable.

Based on the ACF functions we assume a daily and a weekly seasonality. One time series of SET1 and SET2 was picked as an example and the utilization

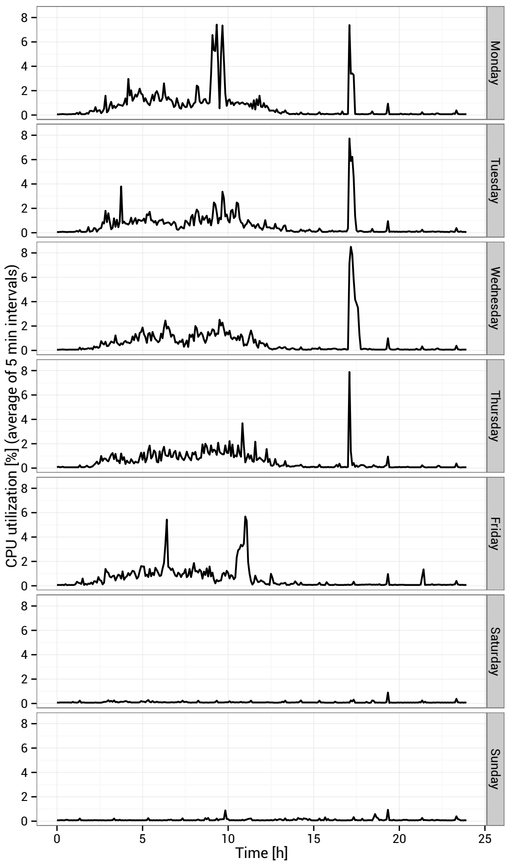


Figure E.3: One week sample of a randomly picked time series of SET1

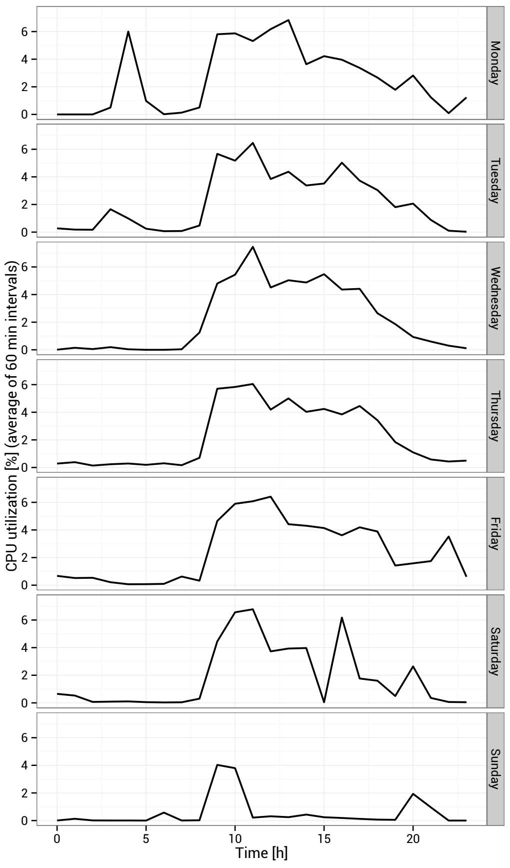


Figure E.4: One week sample of a randomly picked time series of SET2

of an arbitrary chosen week was plotted in Figures E.3 and E.4. All data is normalized to the maximum value of each time series. This data is used to estimate the CPU utilization on a server.

For SET1 a daily seasonality is clearly noticeable. Each day shows the same utilization behavior and on Monday through Thursday there is a batch job at around 17 h. No load exists during the weekend which explains the weekly seasonality in the ACF plots. A similar behavior is noticeable for SET2. The plots do not contain as much noise as for SET1 because of the sampling rate of 60 min instead of 5 min for SET1. However, a daily pattern is clearly noticeable and a reduced service demand during the weekend is noticeable as well.

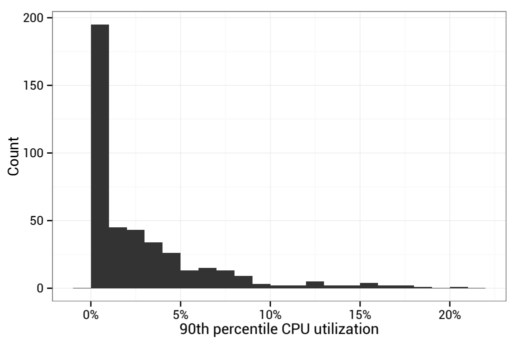


Figure E.5: Histogram of the 90th percentile values over all utilization time series of SET1

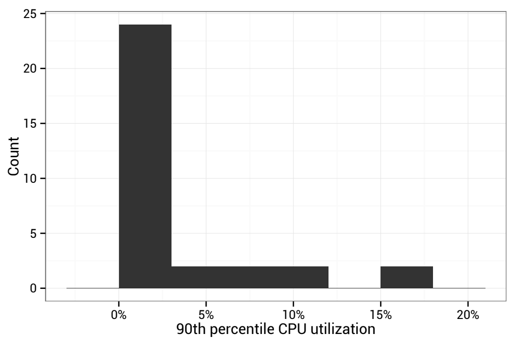


Figure E.6: Histogram of the 90th percentile values over all utilization time series of SET2

Over all 451 utilization time series of SET1 and SET2 we calculated histograms over the 90th percent utilization of each time series. Both are shown in Figures E.5 for SET1 and Figure E.6 for SET2 respectively. The CPU utilization remained below 15 % in most cases. This somewhat confirms other findings like [6] but still is comparable low. One reason might be that enterprise servers are oversized to handle extreme peak load conditions. Another reason might be that services are deployed on dedicated machines for reasons of resource and software isolation without considering their resource footprint so that servers were massively oversized.

We calculated statistical core-metrics over all 451 utilization time series of SET1 and SET2. For each utilization time series we calculated the minimum, maximum, mean, and median utilization value as well as the standard deviation. Over all time series these core-metrics were then aggregated by the mean, median, and 90th percentile functions. Results are reported in Table E.1.

E.2 Workload profile calculation

Workload profiles are calculated as proposed by Speitkamp et al. [86]. For completeness we describe the concrete procedure used in this work. A time series u_l^{raw} is split into a set of seasons S, e.g. of a 24 h duration. Each season

Set	Statistic	Minimum	Maximum	Median	Mean	SD
SET1	Mean	0.04	11.08	0.61	1.12	1.39
SET1	Median	0.02	11.88	0.20	0.49	0.95
SET1	p90	0.03	20.95	1.28	2.80	3.69
SET2	Mean	0.00	7.45	0.15	1.28	2.19
SET2	Median	0.00	5.90	0.09	0.86	1.68
SET2	p90	0.00	16.81	0.32	2.84	4.80

Table E.1: Descriptive statistics over all utilization time series in SET1 and SET2

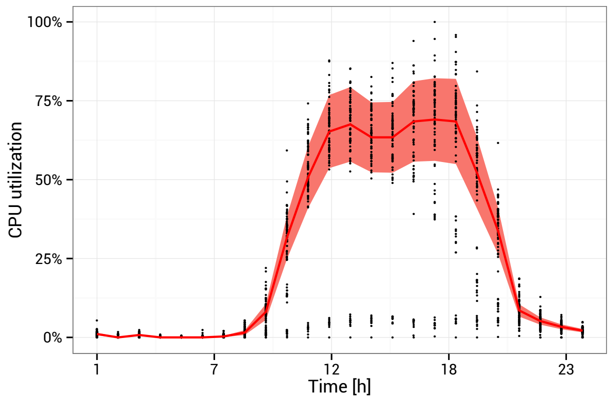


Figure E.7: Calculation of a workload profile based on Equation E.1

consists of τ values indexed by $\{1, \cdots, t, \cdots, \tau\}$. A set U_t contains all values at index t over all seasons S as shown in Equation E.1[86].

$$U_t = \bigcup_{s=0}^{S-1} \{u^{raw}_{(s \cdot \tau + t)}\} \tag{E.1}$$

For each random variable U_t a representative is calculated, e.g. by taking the mean or 95th percentile of the values. This leaves τ representatives for the workload profile. Subsequently, the number of representatives can be down-sampled by aggregating adjacent representatives into one value by taking the maximum or median value. Figure E.7 shows a scatter plot of a time series with the calculated workload profile. In this case weekends were not removed so that some of the data points are very low between 8 h and 21 h.

The following steps were passed through to calculate a workload profile of a given time series:

1. Remove weekends (Saturday and Sunday) from the time series.
2. Eliminate outliers. Outliers are values that are larger than the 99th percentile of the time series values. Each outlier is replaced by a value that is calculated by taking the mean of the $w = 5$ values in front of the outlier (window).
3. Sets U_t are calculated. In contrast to Speitkamp the period is split into τ' equidistant regions resulting in τ' sets U_t. By this, a similar effect as the downsampling done by Speitkamp is achieved.
4. For each set a representative is calculated by an aggregation function. For the workload profiles the mean function is used as aggregation function, resulting in a workload profile of τ' values.
5. The resolution (number of values) of the profile is increased (upscaling) by duplicating values so that a sampling rate of 5 min is achieved. Finally the workload profile is smoothed slightly with a single exponential smoothing average at $\alpha = 0.9$.
6. Gaussian noise is modulated onto the workload profile. Gaussian noise values are chosen out of a range that depends on the standard deviation of the values in sets U_t.

E.3 VM arrival and departure schedules

Texts in this section are based on a previous publication [99].

Schedules define arrival and departure times of VMs as shown in Figure 7.1. Whenever a VM arrives, it is allocated to a server. Correspondingly, whenever a VM departs, it is deallocated from a server. Since VM migrations may be applied, allocation and deallocation servers might not be identical.

Factor	Low	High
Lifetime (h)	1	6
Inter-arrival time (min)	5	20
VM launches	400	500
VM sizes	2^x	x

Table E.2: Factors and levels for schedule configuration sensitivity analysis

A schedule is an instance of a schedule configuration, which comprises the actual factor levels used for generating a schedule instance. A 2^k fully factorial design with $k = 4$ factors (Table E.2) was used to create sixteen different schedule configurations based on the factors that exerted the highest impact on placement controller efficiency. These schedules are of synthetic nature and facilitate screening of significant factors:

- **Lifetime** based on a neg-exponential distribution (bars in Figure 7.1).
- **Inter-arrival time** based on a neg-exponential distribution (time difference between the arrival times of two neighboring launches in Figure 7.1).
- **Launches** as the total number of arriving VMs (total number of bars in Figure 7.1)
- **Sizes** chosen by uniformly distributed random numbers to describe the VMs' resource demand that is either integral with all demands exponents of two, or fractional.

For each of the 16 schedule configurations, 25 schedule instances were created and benchmarked with 9 different placement controllers: First-Fit, First-Fit-Demand, Best-Fit, Best-Fit-Demand, Worst-Fit, Worst-Fit-Demand, Dot-Product, Dot-Product-Demand, and Random. A detailed description of each controller can be found in Chapter 7. This resulted in a total of $16 \cdot 25 \cdot 9 = 3600$ runs.

The simulation framework described in Appendix A provided the necessary infrastructure for conducting all simulation runs. Every simulation run yielded two metrics based on the specific schedule used: average $\overline{\mathrm{SD}}$ and peak server demand $\lceil \mathrm{SD} \rceil$. For example, $\overline{\mathrm{SD}}_{FF}$ is the average server demand for the First-Fit (FF) controller. Calculations are based on Equations 1.1, 1.2.

Allocation efficiency is the preferred performance metric for placement controllers as it denotes underutilized server capacities. Given the theoretical lower bound $\overline{\mathrm{LB}}$ on the server demand for a schedule instance, the efficiency of e.g. the First-Fit controller is then calculated by $\overline{SD}_{FF}/\overline{\mathrm{LB}}$.

Calculations for lower bounds are based on the notation introduced in Section 1.3. For **reservation-based controllers** we extract arrival and departure queries from a stream of VM allocation and deallocation requests as shown in Figure 7.1. VM resource reservations are defined by r_{jk} over $k \in K$ resources and VMs $j \in J$. Server capacity s_k is identical for all servers. $z_{jn} = 1$ if VM j is active after allocation or deallocation query n out of ν queries.

For **demand-based controllers** lower bounds are calculated based on minimum resource utilization d_{ik} since the last query instead of using VM reservations r_{jk}.

A lower bound $\mathrm{LB}(n, k)$ on **maximum server demand** between queries n and $n+1$ for resource k is calculated by Equation E.2. It adds up the resource demands of all active VMs at query n and divides it by the server capacity.

$$\mathrm{LB}(n,k) = \left\lceil \frac{1}{s_k} \sum_{j \in J} z_{jn} r_{jk} \right\rceil \tag{E.2}$$

A lower bound over all queries and resources is then calculated by Equation E.3. It determines the maximum lower bound over all resources and queries.

$$\widehat{\mathrm{LB}} = \max \left(\bigcup_{n=1}^{\nu-1} \bigcup_{k \in K} \mathrm{LB}(n,k) \right) \tag{E.3}$$

A lower bound for the **average server demand** weights the lower bound on maximum server demand for each query by the time delta to the subsequent query. Summing up all weighted lower bounds and dividing the sum by the

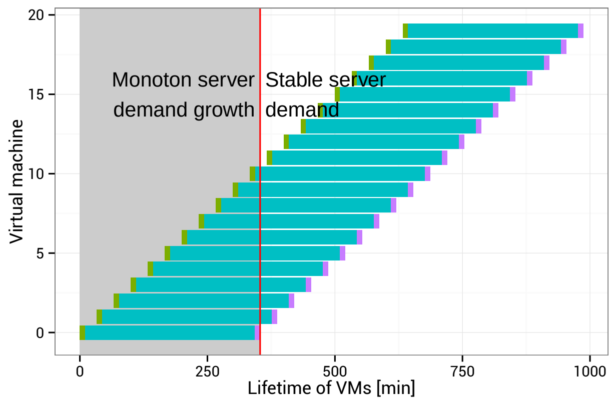


Figure E.8: Peak server demand in a idealized schedule

total time delta from query 1 to ν results in the lower bound on average server demand as shown in Equation E.4.

$$\overline{\text{LB}} = \frac{\sum_{n=1}^{\nu-1} \max_{k \in K} \Big(\text{LB}(n,k) \cdot (a_{n+1} - a_n) \Big)}{a_\nu - a_1} \tag{E.4}$$

Initial simulations used schedule configurations with 40 and 80 VM launches. Contrary to our expectations, the factor VM launches had a significant effect on controller efficiency. Upon closer inspection, the synthetic schedule instances allocated most of the VMs in the beginning of the schedule, with VMs having a lifetime covering most of the schedule duration. In other words, the number of active VMs grew monotonically, explaining the significance of the factor VM launches.

Such a schedule does not reflect the aforementioned stream of allocation and deallocation requests data centers experience. A more realistic, yet idealized schedule is shown in Figure E.8 with interwoven VM allocations and deallocations. The total number of active VMs and the server demand grow monotonically over the first 300 min. After that, server demand stabilizes as VMs get deallocated and new ones get allocated. Therefore, we increased the minimum number of VMs in a schedule to 400.

A dedicated ANOVA analysis for each placement controller found lifetime and inter-arrival time as well as their interaction effect significant in all cases, with levels of $p < 0.001$. In rare cases, the factor launches was significant, with $p < 0.01$. Q-Q plots and residual plots showed no abnormalities.

A TukeyHSD test found no differences *within* each of the two groups of reservation and demand based controllers. Differences *between* the groups were found. In all cases, demand-based significantly outperformed reservation-based controllers. Still, demand-based controllers require dynamic controllers to avoid server overload situations and can, therefore, not be considered as a clear winner.

Appendix F

KMControl and TControl parametrization

Texts in this chapter are based on a previous publication [99].

A design of experiments (DoE) [56] was chosen to find a good parametrization for KMControl and TControl controllers used in Chapter 7. ANOVA and contour plots were used to determine parametrization that minimize the average server demand while keeping migrations and service level violations in a desired operation range.

The simulation framework as described in Appendix A.2 was used. Initially, all VMs are allocated to the servers in a round robin approach were VM with index i is assigned to server $i \, mod \, n$ with n servers. Simulations were conducted for a scenario size of 390 VMs and 90 servers. The number of VMs in the infrastructure did not change during a simulation.

Metrics returned for each simulation were: migration count, server demand, service level violations T^v (number of simulation intervals where server load was above 100 %), and T^a (total number of simulation intervals). A service level is calculated by $1 - {}^{T^v}/_{T^a}$. It is important to understand that calculated service level differs from service levels reported for experiments as explained in Section A.2.

	KMControl		TControl	
Factor	Low	High	Low	High
Overload Threshold T_o	80	95	80	95
Underload Threshold T_u	10	40	10	40
Executiong Interval I (secs)	60	900	60	900
k value	10	100	50	200
m value	8	80	-	-
α	-	-	0.05	0.1

Table F.1: Factors and levels for KM- and TControl parametrization

Simulations were conducted according to a 2^k fully factorial design. Factor levels are shown in Table F.1 for both controllers. Each factor is one controller parameter as explained in Section 7.4. Each treatment was replicated 20 times with different CPU workload traces. In total, 640 simulations were conducted for each controller, discarding invalid factor level combinations for KMControl where $k < m$.

The desired operation range of both controllers was defined based on our experience with European data center operators as follows: Service level above 95 %, minimal number of VM migrations, and minimal average server demand. The following analysis steps are performed on both controllers with respect to each of the metrics: violations, migrations, and servers.

1. Construct a full ANOVA model that contains a combination of all factors.
2. Check Q-Q and residual plots for the full model.
3. Construct a reduced ANOVA model that contains significant factors only.
4. Check Q-Q and residual plots for the reduced model.
5. Create contour plots of all interactions effects.

Exemplary contour plots for the KMControl controller are shown in Figures F.1, F.2, and F.3.

F.1 Results on KMControl

Initial ANOVA analysis over all simulations included all factors and their interactions. All factors were coded to design variables in a range of $[-1, 1]$. Data was log-transformed as initial results indicated heteroscedasticity in the residual plots and deviations from the normality assumption. Some factors were found to be significant at $p < 0.05$. A second linear ANOVA model for each target variable was constructed based on significant factors only. QQ-plots of the model residuals indicated a normal distribution with slight deviation in the tails. Scatter plots of residuals vs. fitted values did not show any patterns or heteroscedasticity. Contour plots were generated based on the linear ANOVA models to find a global optimal parametrization. Based on this plots we gained a number of insights:

1. Server demand can be decreased by choosing a large T_o, a large T_u, and a large m in conjunction with a small k. It is independent to the interval length I.
2. VM migrations can be decreased by increasing T_o, m, and I. However, k and T_u should be small. Especially the demand for a small T_u is in contradiction with the goal to decrease average server demand with a high T_u. We chose the highest level for T_u so that the migration constraint is satisfied in order to minimize server demand.
3. Violations can be decreased by increasing T_o, I, and m while decreasing T_u, and k. These requirements are closely aligned with the requirements to achieve a low number of migrations.

Based on the contour plots, we determined a controller parametrization with $T_o = 95$, $T_u = 27$, $k = m = 65$, and $I = 750$.

F.2 Results on TControl

The same analysis approach as for the KMControl controller was chosen. Based on the contour plots for the TControl controller we gained the following insights:

1. Server demand can be decreased by increasing both T_o and T_u thresholds and by decreasing factors k and I.
2. Migrations can be minimized by increasing T_o, k and I while T_U should be minimized. Increasing T_o and k is in contradiction with the goals to minimize server demand. We choose values so that average server demand is minimized and migrations remain within the desired operation range.
3. Violations are minimized by decreasing all factors while T_o and T_u are the strongest contributors.

Based on the contour plots, we determined a controller parametrization with $T_o = 90$, $T_u = 25$, $k = 170$, and $I = 400$.

F.3 Differences KMControl and TControl

We conducted additional simulations to compare the performance of both controllers. Both were configured with the settings of the previous analysis. Comparison is based on 20 simulations with a different set of workload traces. Key-metrics were: server demand, migrations, and violations. For each metric the mean, medium, maximum, minimum, first-, and third-quartile was calculated.

For server demand, we found TControl to outperform KMControl significantly based on a two-sided t-test at $p = 0.007529$. In all scenarios TControl was better regarding server demand. Both controllers performed equally good for migrations while TControl has a slight tendency to trigger more migrations.

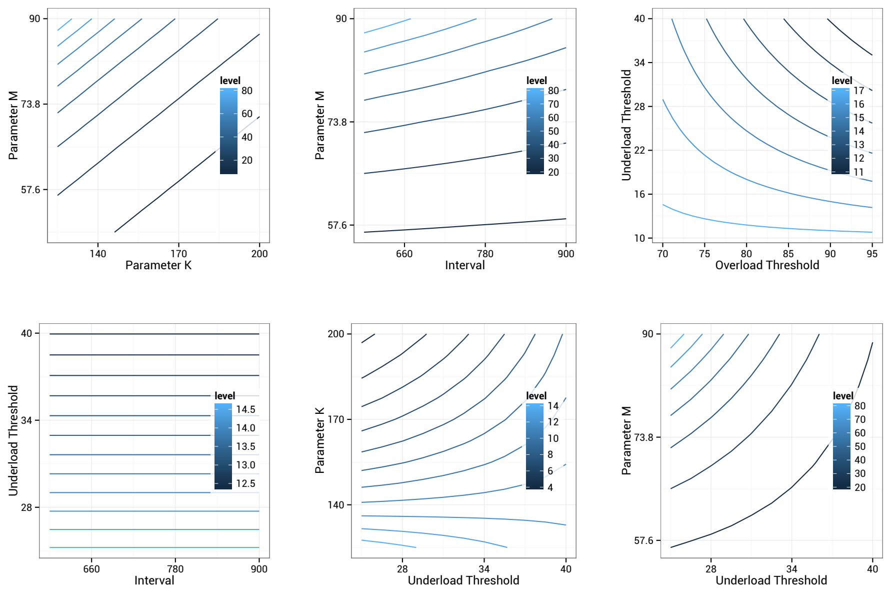


Figure F.1: Effect of parameter setting on server demand for KMControl

A two-sided t-test found significant differences at $p = 0.01888$. For a scenario with 100 VMs over 20 simulation runs, KMControl performed 136 migrations on median vs. 155 migrations for TControl. We could not find significant differences for violations between both controllers with $p = 0.4264$.

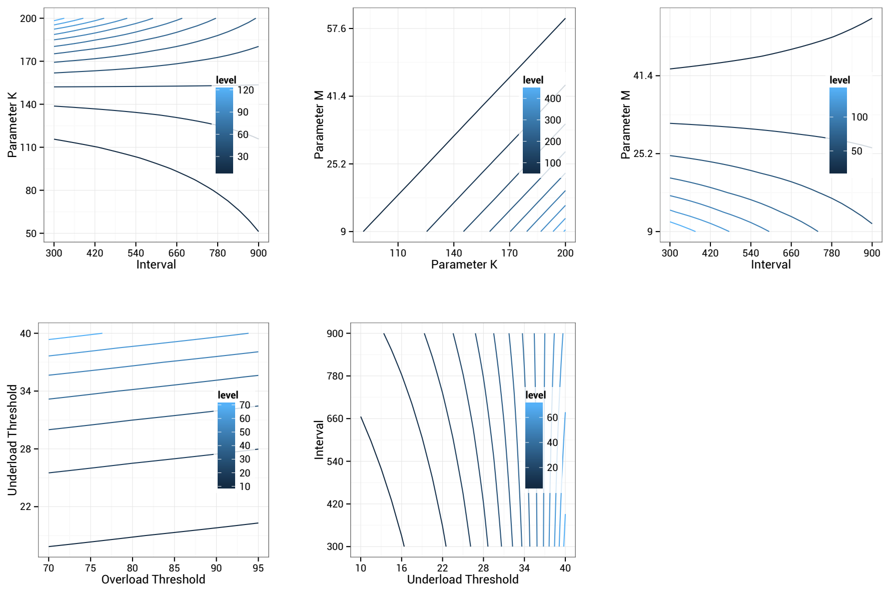


Figure F.2: Effect of parameter setting on migrations for KMControl

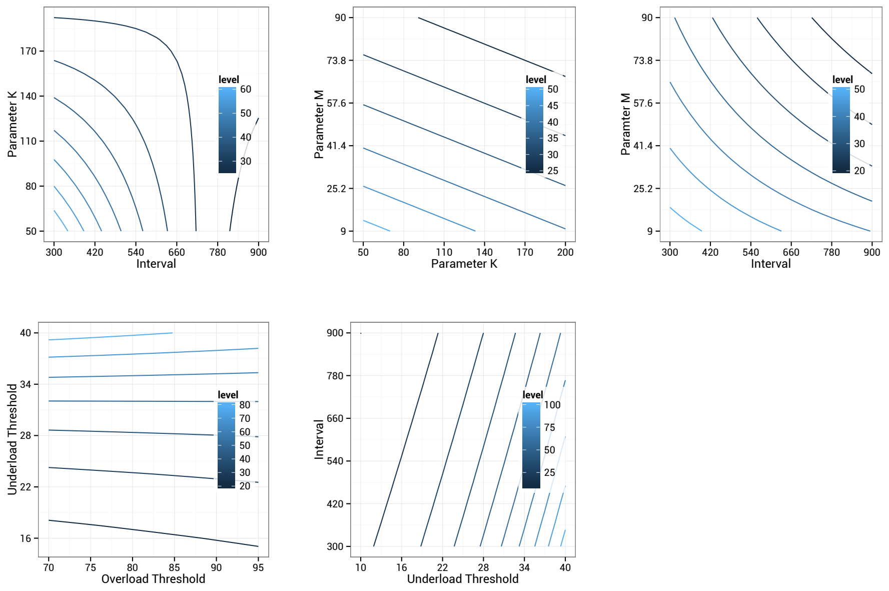


Figure F.3: Effect of parameter setting on violations for KMControl

Bibliography

[1] S. Akoush, R. Sohan, A. Rice, A. Moore, and A. Hopper. Predicting the performance of virtual machine migration. In *Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS), 2010 IEEE International Symposium on*, pages 37–46, Aug 2010. doi: 10.1109/MASCOTS.2010.13.

[2] A. Andrzejak, M. Arlitt, and J. Rolia. Bounding the resource savings of utility computing models. Technical report, HP Laboratories, 2002.

[3] D. Ardagna, B. Panicucci, M. Trubian, and L. Zhang. Energy-aware autonomic resource allocation in multitier virtualized environments. *Services Computing, IEEE Transactions on*, 5(1):2–19, Jan 2012. ISSN 1939-1374. doi: 10.1109/TSC.2010.42.

[4] S. Aulbach, D. Jacobs, A. Kemper, and M. Seibold. A comparison of flexible schemas for software as a service. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 881–888, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-551-2. doi: 10.1145/1559845.1559941. URL `http://doi.acm.org/10.1145/1559845.1559941`.

[5] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37(5):164–177, Oct. 2003. ISSN 0163-5980. doi: 10.1145/1165389.945462. URL `http://doi.acm.org/10.1145/1165389.945462`.

[6] L. A. Barroso and U. Hölzle. The case for energy-proportional computing. *IEEE Computer*, 40, 2007. URL `http://www.computer.org/portal/site/computer/index.jsp?pageID=computer_level1&path=computer/homepage/Dec07&file=feature.xml&xsl=article.xsl`.

[7] A. Beitch, B. Liu, T. Yung, R. Griffith, A. Fox, and D. A. Patterson. Rain: A workload generation toolkit for cloud computing applications. Technical report, Berkeley RAD Lab, 2010.

[8] A. Beloglazov and R. Buyya. Adaptive threshold-based approach for energy-efficient consolidation of virtual machines in cloud data centers. In *Proceedings of the 8th International Workshop on Middleware for Grids, Clouds and e-Science*, MGC '10, pages 4:1–4:6, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0453-5. doi: 10.1145/1890799.1890803. URL `http://doi.acm.org/10.1145/1890799.1890803`.

[9] M. Bichler, T. Setzer, and B. Speitkamp. Capacity planning for virtualized servers. In *Workshop on Information Technologies and Systems (WITS), Milwaukee, Wisconsin, USA*, volume 1. sn, 2006.

[10] M. Bichler, T. Setzer, and B. Speitkamp. Provisioning of resources in a data processing system for services requested. *European Patent No. PCT/EP2007/063361*, 2006.

[11] S. Blanas, Y. Li, and J. M. Patel. Design and evaluation of main memory hash join algorithms for multi-core cpus. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 37–48, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0661-4. doi: 10.1145/1989323.1989328. URL `http://doi.acm.org/10.1145/1989323.1989328`.

[12] N. Bobroff, A. Kochut, and K. Beaty. Dynamic placement of virtual machines for managing sla violations. In *Integrated Network Management, 2007. IM '07. 10th IFIP/IEEE International Symposium on*, pages 119–128, May 2007. doi: 10.1109/INM.2007.374776.

[13] S. Bobrowski. Optimal multitenant designs for cloud apps. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 654–659, July 2011. doi: 10.1109/CLOUD.2011.98.

[14] C. Bodenstein, G. Schryen, and D. Neumann. Energy-aware workload management models for operation cost reduction in data centers. *European Journal of Operational Research*, 222(1):157 – 167, 2012. ISSN 0377-2217. doi: http://dx.doi.org/10.1016/j.ejor.2012.04.005. URL `http://www.sciencedirect.com/science/article/pii/S0377221712002810`.

[15] N. Calcavecchia, O. Biran, E. Hadad, and Y. Moatti. Vm placement strategies for cloud scenarios. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 852–859, June 2012. doi: 10.1109/CLOUD.2012.113.

[16] S. Casolari and M. Colajanni. Short-term prediction models for server management in internet-based contexts. *Decision Support Systems*, 48(1):212 – 223, 2009. ISSN 0167-9236. doi: http://dx.doi.org/10.1016/j.dss.2009.07.014. URL `http://www.sciencedirect.com/science/article/pii/S0167923609001778`. Information product markets.

[17] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2): 4:1–4:26, June 2008. ISSN 0734-2071. doi: 10.1145/1365815.1365816. URL `http://doi.acm.org/10.1145/1365815.1365816`.

[18] W. chun Feng, X. Feng, and R. Ge. Green supercomputing comes of age. *IT Professional*, 10(1):17–23, Jan 2008. ISSN 1520-9202. doi: 10.1109/MITP.2008.8.

[19] E. Coffman, Jr., M. Garey, and D. Johnson. Dynamic bin packing. *SIAM Journal on Computing*, 12(2):227–258, 1983. doi: 10.1137/0212014. URL `http://dx.doi.org/10.1137/0212014`.

[20] E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson. Approximation algorithms for np-hard problems. In D. S. Hochbaum, editor, –, chapter Approximation Algorithms for Bin Packing: A Survey, pages 46–93. PWS Publishing Co., Boston, MA, USA, 1997. ISBN 0-534-94968-1. URL `http://dl.acm.org/citation.cfm?id=241938.241940`.

[21] R. Cole, F. Funke, L. Giakoumakis, W. Guy, A. Kemper, S. Krompass, H. Kuno, R. Nambiar, T. Neumann, M. Poess, K.-U. Sattler, M. Seibold, E. Simon, and F. Waas. The mixed workload ch-benchmark. In *Proceedings of the Fourth International Workshop on Testing Database Systems*, DBTest '11, pages 8:1–8:6, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0655-3. doi: 10.1145/1988842.1988850. URL `http://doi.acm.org/10.1145/1988842.1988850`.

[22] G. Dasgupta, A. Sharma, A. Verma, A. Neogi, and R. Kothari. Workload management for power efficiency in virtualized data centers. *Commun. ACM*, 54(7):131–141, July 2011. ISSN 0001-0782. doi: 10.1145/1965724.1965752. URL `http://doi.acm.org/10.1145/1965724.1965752`.

[23] S. De Chaves, R. Uriarte, and C. Westphall. Toward an architecture for monitoring private clouds. *Communications Magazine, IEEE*, 49(12):130–137, December 2011. ISSN 0163-6804. doi: 10.1109/MCOM.2011.6094017.

[24] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. A. Wood. Implementation techniques for main memory database systems. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, SIGMOD '84, pages 1–8, New York, NY, USA, 1984. ACM. ISBN 0-89791-128-8. doi: 10.1145/602259.602261. URL `http://doi.acm.org/10.1145/602259.602261`.

[25] G. Dhiman, G. Marchetti, and T. Rosing. vgreen: A system for energy efficient computing in virtualized environments. In *Proceedings of the 2009 ACM/IEEE International Symposium on Low Power Electronics and Design*, ISLPED '09, pages 243–248, New York, NY, USA, 2009.

ACM. ISBN 978-1-60558-684-7. doi: 10.1145/1594233.1594292. URL `http://doi.acm.org/10.1145/1594233.1594292`.

[26] X. Fan, W.-D. Weber, and L. A. Barroso. Power provisioning for a warehouse-sized computer. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 13–23, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-706-3. doi: 10.1145/1250662.1250665. URL `http://doi.acm.org/10.1145/1250662.1250665`.

[27] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner. Sap hana database: Data management for modern business applications. *SIGMOD Rec.*, 40(4):45–51, Jan. 2012. ISSN 0163-5808. doi: 10.1145/2094114.2094126. URL `http://doi.acm.org/10.1145/2094114.2094126`.

[28] D. Filani, J. He, S. Gao, M. Rajappa, A. Kumar, P. Shah, and R. Nagappan. Dynamic data center power management: Trends, issues, and solutions. Technical Report 1, Intel Technology Journal, 2008.

[29] H. Frenk, J. Csirik, M. Labbé, and S. Zhang. On the multidimensional vector bin packing. *University of Szeged. Acta Cybernetica*, pages 361–369, 1990.

[30] F. Funke, A. Kemper, S. Krompass, H. Kuno, R. Nambiar, T. Neumann, A. Nica, M. Poess, and M. Seibold. Metrics for measuring the performance of the mixed workload ch-benchmark. In R. Nambiar and M. Poess, editors, *Topics in Performance Evaluation, Measurement and Characterization*, volume 7144 of *Lecture Notes in Computer Science*, pages 10–30. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-32626-4. doi: 10.1007/978-3-642-32627-1_2. URL `http://dx.doi.org/10.1007/978-3-642-32627-1_2`.

[31] L. George. *HBase: The Definitive Guide*. O'Reilly Media, 2011.

[32] D. Gmach, J. Rolia, L. Cherkasova, and A. Kemper. Workload analysis and demand prediction of enterprise data center applications. In *Workload Characterization, 2007. IISWC 2007. IEEE 10th International Symposium on*, pages 171–180, Sept 2007. doi: 10.1109/IISWC.2007.4362193.

[33] D. Gmach, J. Rolia, and L. Cherkasova. Satisfying service level objectices in a self-managing resource pool. In *Self-Adaptive and Self-Organizing Systems, 2009. SASO '09. Third IEEE International Conference on*, pages 243–253, Sept 2009. doi: 10.1109/SASO.2009.27.

[34] D. Gmach, J. Rolia, L. Cherkasova, and A. Kemper. Resource pool management: Reactive versus proactive or let's be friends. *Computer Networks*, 53(17):2905–2922, 2009. ISSN 1389-1286. doi: http://dx.doi.org/10.1016/j.comnet.2009.08.011. URL `http://www.sciencedirect.com/science/article/pii/S1389128609002655`. Virtualized Data Centers.

[35] H. Goudarzi and M. Pedram. Multi-dimensional sla-based resource allocation for multi-tier cloud computing systems. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 324–331, July 2011. doi: 10.1109/CLOUD.2011.106.

[36] P. Graubner, M. Schmidt, and B. Freisleben. Energy-efficient management of virtual machines in eucalyptus. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 243–250, July 2011. doi: 10.1109/CLOUD.2011.26.

[37] A. Gulati, A. Holler, M. Ji, G. Shanmuganathan, C. Waldspurger, and X. Zhu. Vmware distributed resource management: Design, implementation, and lessons learned. *VMware Technical Journal*, 1(1):45–64, 2012.

[38] J. Hall, J. Hartline, A. R. Karlin, J. Saia, and J. Wilkes. On algorithms for efficient data migration. In *Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '01, pages 620–629,

Philadelphia, PA, USA, 2001. Society for Industrial and Applied Mathematics. ISBN 0-89871-490-7. URL `http://dl.acm.org/citation.cfm?id=365411.365549`.

[39] J. D. Hamilton. *Time Series Analysis*. Princeton University Press, 1994. ISBN 0691042896.

[40] A. Hios and T. Ulichnie. Top 10 data center business management priorities for 2013 about the uptime institute network. Technical report, Uptime Institute, 2013.

[41] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, Aug. 1978. ISSN 0001-0782. doi: 10.1145/359576.359585. URL `http://doi.acm.org/10.1145/359576.359585`.

[42] C. Ingle. Beyond organisational boundaries: Answering the enterprise computing challenge. Technical report, IDC, 2009.

[43] Z. Ivkovic and E. Lloyd. Fully dynamic algorithms for bin packing: Being (mostly) myopic helps. *SIAM Journal on Computing*, 28(2):574–611, 1998. doi: 10.1137/S0097539794276749. URL `http://dx.doi.org/10.1137/S0097539794276749`.

[44] D. Kegel. The c10k problem, 2014. URL `http://www.kegel.com/c10k.html`.

[45] A. Kemper and T. Neumann. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. *2014 IEEE 30th International Conference on Data Engineering*, 0:195–206, 2011. doi: http://doi.ieeecomputersociety.org/10.1109/ICDE.2011.5767867.

[46] S. Kikuchi and Y. Matsumoto. Performance modeling of concurrent live migration operations in cloud computing systems using prism probabilistic model checker. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 49–56, July 2011. doi: 10.1109/CLOUD.2011.48.

[47] C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, A. D. Nguyen, N. Satish, J. Chhugani, A. Di Blas, and P. Dubey. Sort vs. hash revisited: Fast join implementation on modern multi-core cpus. *Proc. VLDB Endow.*, 2(2): 1378–1389, Aug. 2009. ISSN 2150-8097. doi: 10.14778/1687553.1687564. URL `http://dx.doi.org/10.14778/1687553.1687564`.

[48] B. Knafla. Introduction to behavior trees, 2 2011. URL `http://www.altdevblogaday.com/2011/02/24/introduction-to-behavior-trees/`.

[49] S. Krompass, H. Kuno, J. L. Wiener, K. Wilkinson, U. Dayal, and A. Kemper. Managing long-running queries. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '09, pages 132–143, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-422-5. doi: 10.1145/1516360.1516377. URL `http://doi.acm.org/10.1145/1516360.1516377`.

[50] J. Kroßand A. Wolke. Cloudburst - simulating workload for iaas clouds. In *Cloud Computing (CLOUD), 2014 IEEE 7th International Conference on*, 2014.

[51] B. Li, J. Li, J. Huai, T. Wo, Q. Li, and L. Zhong. Enacloud: An energy-saving application live placement approach for cloud computing environments. In *Cloud Computing, 2009. CLOUD '09. IEEE International Conference on*, pages 17–24, Sept 2009. doi: 10.1109/CLOUD.2009.72.

[52] X. Li, A. Ventresque, N. Stokes, J. Thorburn, and J. Murphy. ivmp: An interactive vm placement algorithm for agile capital allocation. In *Proceedings of the 2013 IEEE Sixth International Conference on Cloud Computing*, CLOUD '13, pages 950–951, Washington, DC, USA, 2013. IEEE Computer Society. ISBN 978-0-7695-5028-2. doi: 10.1109/CLOUD.2013.4. URL `http://dx.doi.org/10.1109/CLOUD.2013.4`.

[53] H. Liu. A measurement study of server utilization in public clouds. In *Dependable, Autonomic and Secure Computing (DASC), 2011 IEEE*

Ninth International Conference on, pages 435–442, Dec 2011. doi: 10.1109/DASC.2011.87.

[54] K. Maruyama, S. Chang, and D. Tang. A general packing algorithm for multidimensional resource requirements. *International Journal of Computer & Information Sciences*, 6(2):131–149, 1977. ISSN 0091-7036. doi: 10.1007/BF00999302. URL `http://dx.doi.org/10.1007/BF00999302`.

[55] K. Mills, J. Filliben, and C. Dabrowski. An efficient sensitivity analysis method for large cloud simulations. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 724–731, July 2011. doi: 10.1109/CLOUD.2011.50.

[56] D. C. Montgomery. *Design and Analysis of Experiments*. John Wiley & Sons, 2006. ISBN 0470088109.

[57] H. Mühe, A. Kemper, and T. Neumann. How to efficiently snapshot transactional data: Hardware or software controlled? In *Proceedings of the Seventh International Workshop on Data Management on New Hardware*, DaMoN '11, pages 17–26, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0658-4. doi: 10.1145/1995441.1995444. URL `http://doi.acm.org/10.1145/1995441.1995444`.

[58] M. Mutsuzaki. Thrift java servers compared, 2013. URL `https://github.com/m1ch1/mapkeeper/wiki/Thrift-Java-Servers-Compared`.

[59] NASCIO. State cio priorities for 2013. Technical report, NASCIO, 2013.

[60] M. Nelson, B.-H. Lim, and G. Hutchins. Fast transparent migration for virtual machines. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 25–25, Berkeley, CA, USA, 2005. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1247360.1247385`.

[61] K. Parent. Consolidation improves it's capacity utilization. Technical report, Court Square Data Group, 2005.

[62] N. B. V. Partner. Future of cloud computing survey. Technical report, North Bridge Venture Partners, 2011.

[63] C. Peng, M. Kim, Z. Zhang, and H. Lei. Vdn: Virtual machine image distribution network for cloud data centers. In *INFOCOM, 2012 Proceedings IEEE*, pages 181–189, March 2012. doi: 10.1109/INFCOM.2012.6195556.

[64] R. Pike. Go at google: Language design in the service of software engineering, 2012. URL `http://talks.golang.org/2012/splash.article`.

[65] S. Piramuthu. On learning to predict web traffic. *Decision Support Systems*, 35(2):213 – 229, 2003. ISSN 0167-9236. doi: http://dx.doi.org/10.1016/S0167-9236(02)00107-0. URL `http://www.sciencedirect.com/science/article/pii/S0167923602001070`. Web Data Mining.

[66] H. Plattner. A common database approach for oltp and olap using an in-memory column database. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 1–2, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-551-2. doi: 10.1145/1559845.1559846. URL `http://doi.acm.org/10.1145/1559845.1559846`.

[67] V. Radulovic. Recommendations for tier i energy star computer specification. Technical report, United States Environmental Protection Agency, Pittsburgh, PA, 2011.

[68] M. S. Rehman, M. Hammoud, and M. F. Sakr. Votus: A flexible and scalable monitoring framework for virtualized clusters. *In Proceedings of The 3rd International Conference on Cloud Computing and Science (CloudCom 2011), Athens, Greece*, pages 1–4, 2011.

[69] K. Ren, J. López, and G. Gibson. Otus: Resource attribution in data-intensive clusters. In *Proceedings of the Second International Workshop on MapReduce and Its Applications*, MapReduce '11, pages 1–8,

New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0700-0. doi: 10.1145/1996092.1996094. URL `http://doi.acm.org/10.1145/1996092.1996094`.

[70] J. Rolia, X. Zhu, M. Arlitt, and A. Andrzejak. Statistical service assurances for applications in utility grid environments. *Performance Evaluation*, 58(2-3):319–339, 2004. ISSN 0166-5316. doi: http://dx.doi.org/10.1016/j.peva.2004.07.015. URL `http://www.sciencedirect.com/science/article/pii/S0166531604000793`. Distributed Systems Performance.

[71] J. Rolia, L. Cherkasova, M. Arlitt, and A. Andrzejak. A capacity management service for resource pools. In *Proceedings of the 5th International Workshop on Software and Performance*, WOSP '05, pages 229–237, New York, NY, USA, 2005. ACM. ISBN 1-59593-087-6. doi: 10.1145/1071021.1071047. URL `http://doi.acm.org/10.1145/1071021.1071047`.

[72] B. Sanden. Coping with java threads. *Computer*, 37(4):20–27, April 2004. ISSN 0018-9162. doi: 10.1109/MC.2004.1297297.

[73] M. Seibold, A. Kemper, and D. Jacobs. Strict slas for operational business intelligence. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 25–32, July 2011. doi: 10.1109/CLOUD.2011.22.

[74] M. Seibold, A. Wolke, M. Albutiu, M. Bichler, A. Kemper, and T. Setzer. Efficient deployment of main-memory dbms in virtualized data centers. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 311–318, June 2012. doi: 10.1109/CLOUD.2012.13.

[75] S. S. Seiden. On the online bin packing problem. *J. ACM*, 49(5):640–671, Sept. 2002. ISSN 0004-5411. doi: 10.1145/585265.585269. URL `http://doi.acm.org/10.1145/585265.585269`.

[76] S. Seltzsam, D. Gmach, S. Krompass, and A. Kemper. Autoglobe: An automatic administration concept for service-oriented database ap-

plications. In *Data Engineering, 2006. ICDE '06. Proceedings of the 22nd International Conference on*, pages 90–90, April 2006. doi: 10.1109/ICDE.2006.26.

[77] T. Setzer and M. Bichler. Using matrix approximation for high-dimensional discrete optimization problems: Server consolidation based on cyclic time-series data. *European Journal of Operational Research*, 227(1):62–75, 2013. ISSN 0377-2217. doi: http://dx.doi.org/10.1016/j.ejor.2012.12.005. URL `http://www.sciencedirect.com/science/article/pii/S0377221712009368`.

[78] T. Setzer and A. Wolke. Virtual machine re-assignment considering migration overhead. In *Network Operations and Management Symposium (NOMS), 2012 IEEE*, pages 631–634, April 2012. doi: 10.1109/NOMS.2012.6211973.

[79] T. Setzer, M. Bichler, and B. Speitkamp. Capacity management for virtualized servers. In *INFORMS Workshop on Information Technologies and Systems (WITS)*, Milwaukee, USA, 2006.

[80] B. Sigoure. Lessons learned from opentsdb, 2012. URL `http://www.cloudera.com/resource/hbasecon-2012-lessons-learned-from-opentsdb/`.

[81] B. Sigoure. Opentsdb tired of 10 + year old monitoring systems? OSCON, 2012.

[82] E. Smith. jvm-serializers, 2012. URL `https://github.com/eishay/jvm-serializers/wiki/Staging-Results`.

[83] J. H. Son and M. H. Kim. An analysis of the optimal number of servers in distributed client/server environments. *Decision Support Systems*, 36(3):297 – 312, 2004. ISSN 0167-9236. doi: http://dx.doi.org/10.1016/S0167-9236(02)00142-2. URL `http://www.sciencedirect.com/science/article/pii/S0167923602001422`.

[84] A. A. Soror, U. F. Minhas, A. Aboulnaga, K. Salem, P. Kokosielis, and S. Kamath. Automatic virtual machine configuration for database workloads. *ACM Trans. Database Syst.*, 35(1):7:1–7:47, Feb. 2008. ISSN 0362-5915. doi: 10.1145/1670243.1670250. URL `http://doi.acm.org/10.1145/1670243.1670250`.

[85] B. Sotomayor, R. S. Montero, I. Llorente, and I. Foster. Virtual infrastructure management in private and hybrid clouds. *Internet Computing, IEEE*, 13(5):14–22, Sept 2009. ISSN 1089-7801. doi: 10.1109/MIC.2009.119.

[86] B. Speitkamp and M. Bichler. A mathematical programming approach for server consolidation problems in virtualized data centers. *Services Computing, IEEE Transactions on*, 3(4):266–278, Oct 2010. ISSN 1939-1374. doi: 10.1109/TSC.2010.25.

[87] 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*, HotPower'08, pages 10–10, Berkeley, CA, USA, 2008. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1855610.1855620`.

[88] A. Stage. *A Study of Resource Allocation Methods in Virtualized Enterprise Data Centres.* Dissertation, Technische Universität München, München, 2013.

[89] M. Stillwell, D. Schanzenbach, F. Vivien, and H. Casanova. Resource allocation algorithms for virtualized service hosting platforms. *Journal of Parallel and Distributed Computing*, 70(9):962–974, 2010. ISSN 0743-7315. doi: http://dx.doi.org/10.1016/j.jpdc.2010.05.006. URL `http://www.sciencedirect.com/science/article/pii/S0743731510000997`.

[90] K. Talwar, R. Ramasubramanian, R. Panigrahy, and U. Wieder. Validating heuristics for virtual machines consolidation. Technical report, Mi-

crosoft Research, 2011. URL `http://research.microsoft.com/apps/pubs/default.aspx?id=144571`.

[91] R. Van Renesse, K. P. Birman, and W. Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Trans. Comput. Syst.*, 21(2):164–206, May 2003. ISSN 0734-2071. doi: 10.1145/762483.762485. URL `http://doi.acm.org/10.1145/762483.762485`.

[92] A. Verma, P. Ahuja, and A. Neogi. pmapper: Power and migration cost aware application placement in virtualized systems. In V. Issarny and R. Schantz, editors, *Middleware 2008*, volume 5346 of *Lecture Notes in Computer Science*, pages 243–264. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-89855-9. doi: 10.1007/978-3-540-89856-6_13. URL `http://dx.doi.org/10.1007/978-3-540-89856-6_13`.

[93] R. Vermeersch. Concurrency in erlang & scala: The actor model, 2009. URL `http://savanne.be/articles/concurrency-in-erlang-scala/`.

[94] W. Voorsluys, J. Broberg, S. Venugopal, and R. Buyya. Cost of virtual machine live migration in clouds: A performance evaluation. In M. Jaatun, G. Zhao, and C. Rong, editors, *Cloud Computing*, volume 5931 of *Lecture Notes in Computer Science*, pages 254–265. Springer Berlin Heidelberg, 2009. ISBN 978-3-642-10664-4. doi: 10.1007/978-3-642-10665-1_23. URL `http://dx.doi.org/10.1007/978-3-642-10665-1_23`.

[95] A. Wolke and C. Pfeiffer. Improving enterprise vm consolidation with high-dimensional load profiles. In *Cloud Engineering (IC2E), 2014 IEEE International Conference on*, 2014.

[96] A. Wolke and D. Srivastav. Monitoring and controlling research experiments in cloud testbeds. In *Proceedings of the 2013 IEEE Sixth International Conference on Cloud Computing*, CLOUD '13, pages

962–963, Washington, DC, USA, 2013. IEEE Computer Society. ISBN 978-0-7695-5028-2. doi: 10.1109/CLOUD.2013.97. URL `http://dx.doi.org/10.1109/CLOUD.2013.97`.

[97] A. Wolke and L. Ziegler. Evaluating dynamic resource allocation strategies in virtualized data centers. In *Cloud Computing (CLOUD), 2014 IEEE 7th International Conference on*, 2014.

[98] A. Wolke, M. Bichler, and T. Setzer. Planning vs. dynamic control: Resource allocation in corporate clouds. *IEEE Transactions on Cloud Computing*, 2014. Accepted.

[99] A. Wolke, T.-A. Boldbaatar, C. Pfeiffer, and M. Bichler. More than bin packing: A large-scale experiment on dynamic resource allocation in iaas clouds. *Under Review*, 2014.

[100] P. W. Wong, F. C. Yung, and M. Burcea. An 8/3 lower bound for online dynamic bin packing. In K.-M. Chao, T.-s. Hsu, and D.-T. Lee, editors, *Algorithms and Computation*, volume 7676 of *Lecture Notes in Computer Science*, pages 44–53. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-35260-7. doi: 10.1007/978-3-642-35261-4_8. URL `http://dx.doi.org/10.1007/978-3-642-35261-4_8`.

[101] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif. Sandpiper: Black-box and gray-box resource management for virtual machines. *Computer Networks*, 53(17):2923 – 2938, 2009. ISSN 1389-1286. doi: http://dx.doi.org/10.1016/j.comnet.2009.04.014. URL `http://www.sciencedirect.com/science/article/pii/S1389128609002035`. Virtualized Data Centers.

[102] K. Ye, X. Jiang, D. Huang, J. Chen, and B. Wang. Live migration of multiple virtual machines with resource reservation in cloud computing environments. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 267–274, July 2011. doi: 10.1109/CLOUD.2011.69.