Final Optimization Model

Project Deliverable D3.4

Ahmed Ali-Eldin, Jakub Krzywda, Ewnetu Bayuh Lakew, Mina Sedeghat, P-O Östberg (UMU), Jörg Domaschka (UULM)

Due date: 30/09/2016
Delivery date: 29/09/2016
<table>
<thead>
<tr>
<th>Dissemination Level</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>X</strong></td>
</tr>
</tbody>
</table>
| **PU**              | Public
| **PP**              | Restricted to other programme participants (including the Commission Services)
| **RE**              | Restricted to a group specified by the consortium (including the Commission Services)
| **CO**              | Confidential, only for members of the consortium (including the Commission Services) |
### Version History

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Change</th>
<th>Author</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>29/08/2016</td>
<td>Initial Document Structure</td>
<td>Ahmed Ali-Eldin (UMU), P-O Östberg (UMU), Jakub Krzywda (UMU)</td>
</tr>
<tr>
<td>0.2</td>
<td>06/09/2016</td>
<td>Initial Document Content</td>
<td>Ahmed Ali-Eldin (UMU)</td>
</tr>
<tr>
<td>0.3</td>
<td>22/09/2016</td>
<td>First version ready for review</td>
<td>Ahmed Ali-Eldin (UMU)</td>
</tr>
<tr>
<td>0.4</td>
<td>22/09/2016</td>
<td>First review</td>
<td>Jörg Domaschka (UULM)</td>
</tr>
<tr>
<td>1.0</td>
<td>28/09/2016</td>
<td>Addressing comments from the first review</td>
<td>Jakub Krzywda (UMU), Ahmed Ali-Eldin (UMU)</td>
</tr>
</tbody>
</table>
EXECUTIVE SUMMARY

This deliverable describes the final version of the optimization model and algorithms implemented in CactoOpt. The model and algorithms include description of the implemented autoscaling algorithms, their integration with the CACTOS toolkits, and related performance results. In addition, the document describes research results obtained within CACTOS.

There are five optimization capabilities of CactoOpt that can be performed on the logical (software) level of data center management: initial placement of virtual machines, migration of virtual machines, shut down of physical machines for energy savings, horizontal scaling, and vertical scaling. Using these four actuators, CactoOpt optimizes the power, performance, and cost tradeoffs of applications running on CACTOS enabled datacenters. The four main actuators enable CactoOpt to optimize for a wide range of scenarios including consolidation, and load balancing.

This document elaborates the advances within CactoOpt since D3.3. This includes the improvements in the optimization models, and a thorough description of the new vertical scaling algorithms, horizontal scaling algorithms, fault-tolerant scheduling algorithms, and power capping and management, as well as, the interplay of all optimization capabilities.
# Table of Contents

## Executive Summary

## Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Figures</td>
<td>III</td>
</tr>
<tr>
<td>Abbreviations</td>
<td>IV</td>
</tr>
</tbody>
</table>

## I. Introduction

## II. New Models

1. Scientific Workload Models
   - Job Characterization
   - Scientific Workloads Evolution

2. Fault Tolerance Optimization Model
   - Network and Power Topology
   - The Reliability Model
   - Approximating the Reliability Value
   - Effect on Optimization

3. Power-Performance Models and Tradeoffs
   - Description
   - DVFS
   - Horizontal vs Vertical Scaling Consolidation Effects
   - Discussion

## III. New Optimization Algorithms

1. Autoscaling
   - Horizontal Scaling Algorithms
   - Vertical Scaling
   - Integration with CactoOpt
   - Evaluation of the Horizontal Scaling Algorithms
   - Evaluation of the Vertical Scaling Models

2. CactoOpt in Action

3. Visualisation of the Results

## IV. Future Plans

---

*Page D3.3 Extended Optimization Model CACTOS*
## List of Figures

**Figure 1** Job characterization on Hopper, Edison, Carver ........................................................... 4

**Figure 2** Seasonality analysis of the workloads ................................................................................... 5

**Figure 3** The difference in power consumption between various CPU frequencies grows with the number of fully utilized cores .................................................................................. 13

**Figure 4** Influence of DVFS on the response time (RT) and power consumption for AMD servers. Dashed .......... 14

**Figure 5** Influence of DVFS (Intel). For concurrent requests reducing CPU frequency causes an increase in the power consumption of an underloaded server .......................................................... 15

**Figure 6** Comparison of horizontal and vertical scaling and the influence of CPU pinning on the response time (RT) and power consumption. ................................................................. 17

**Figure 7** Influence of load balancing strategy on the response time (RT) and power consumption .................. 19

**Figure 8** Capacity allocation and response times for RUBiS under open and closed system models with 0.5s and 1.5s target response times .......................................................................... 27

**Figure 9** Capacity allocation and response times for RUBBoS under open and closed system models with a 0.5s target response time .................................................................................. 27

**Figure 10** Capacity allocation and response times for Olio under open and closed system models with a 0.5s target response time ...................................................................................... 28

**Figure 11** Visualizer interface .................................................................................................................. 30

**Figure 12** Graphs showing statistics on CPU bounds at a step .................................................................... 31

**Figure 13** Graphs showing statistics on the number of empty machines across the time of the simulation .......... 31

**Figure 14** Graph showing the RAM utilization for a given step of the simulation ......................................... 32
**Abbreviations**

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CACTOS</td>
<td>Context-Aware Cloud Topology Optimisation and Simulation</td>
</tr>
<tr>
<td>SLA</td>
<td>Service Level Agreement</td>
</tr>
<tr>
<td>QoS</td>
<td>Quality of Service</td>
</tr>
<tr>
<td>DVFS</td>
<td>Dynamic Frequency Voltage Scaling</td>
</tr>
<tr>
<td>VM</td>
<td>Virtual Machine</td>
</tr>
<tr>
<td>BLO</td>
<td>Business Level Objective</td>
</tr>
<tr>
<td>WAC</td>
<td>Workload Analysis and Classification tool</td>
</tr>
</tbody>
</table>
I. INTRODUCTION

This accompanying report aims to describe the final WP3 deliverable, the CACTOS optimization model that features predictive capabilities. The purpose of this report is to demonstrate how the enhanced model and advanced optimization algorithms support the optimization of a data centre configuration.

Table 1 shows the overview of the optimisation capabilities implemented as features of CactoOpt and indicates if they were included in the final release of the CactoOpt (June 2016) and when they were demonstrated during the reviews. The goal of this work package is to take an application-centric approach to develop and implement models and algorithms to advance infrastructure manageability in datacentres. In this document we describe new results achieved in application performance modelling (task 3.1) where we have analysed traces from large-scale datacenters to complement the work done on the CACTOS use-cases. The aim of this further analysis is to assist us in developing a more generic CactoOpt version suitable for both CACTOS enabled and non-CACTOS enabled datacenters in collaboration with Lawrence Berkeley National Labs (LBNL) in the US, we have analysed traces provided by them from two cluster systems (Hopper and Carver) at the National Energy Research Scientific Computing (NERSC) Center. In addition, we have analysed and characterised energy and power-performance trade-offs for different Request-Response workloads, and for compute bound applications. These results enabled us to develop more advanced optimisation algorithms for task 3.3. Regarding the development and implementation of cloud topology optimization models (task 3.3), we present an enhanced optimization model encompassing advanced autoscaling, both vertical and horizontal, that use the Application Behaviour Predictor developed as part of task 3.2. Finally, working with researchers from Google, we have looked at the problem of how to improve the optimization algorithms to take in to account fault tolerance. In this deliverable, we use excerpts from some of our published papers and some papers submitted or in preparation. The reminder of the document is structured as follows: Section II describes the workloads analysed and the behavioural models extracted, Section III focuses on autoscaling and the predictive capabilities of CactoOpt, as well as, introduces the advanced optimization algorithms produced in Y3. Finally, Section IV presents future plans for CactoOpt beyond the project.


<table>
<thead>
<tr>
<th>Feature</th>
<th>Implemented and Tested</th>
<th>Integrated in the Final Release</th>
<th>Demo at Review</th>
</tr>
</thead>
<tbody>
<tr>
<td>Placement</td>
<td>Yes</td>
<td>Yes</td>
<td>Year 1 &amp; 2</td>
</tr>
<tr>
<td>Migration</td>
<td>Yes</td>
<td>Yes</td>
<td>Year 2</td>
</tr>
<tr>
<td>Shut down Physical Machines</td>
<td>Yes</td>
<td>Yes</td>
<td>Year 2</td>
</tr>
<tr>
<td>Horizontal Scaling</td>
<td>Yes</td>
<td>Yes</td>
<td>Year 2.5</td>
</tr>
<tr>
<td>Vertical Scaling</td>
<td>Yes</td>
<td>No</td>
<td>-</td>
</tr>
</tbody>
</table>
II. NEW MODELS

In this section, we describe two models that have been developed during year 3 of the project and that were used as input to new features in CactoOpt. The first model is a model for scientific workloads from traces of three large clusters. The second is a model for the power-performance trade-offs for various CactoOpt actuation mechanisms. These models are used to understand better how do different optimisation actions compare.

1. SCIENTIFIC WORKLOAD MODELS

Traditionally, scientific applications running at HPC centers have been large monolithic MPI applications which require high bandwidth, low latency interconnects. However, large scientific computations that include high-throughput, data-intensive jobs, and stream-processing are increasingly becoming more common in HPC centers. Many of these applications can benefit from being deployed in a cloud rather than an HPC center. In CACTOS, one of the main use-cases is using one of these scientific application, MolPro. As described in D3.3, we have analysed and characterized MolPro’s resources usage. While this enabled us to design CactoOpt’s optimization algorithms, to be able to generalize our results, we need to study more workloads from large datacentre production environments. In traditional HPC environments, resources are assigned using batch queue schedulers with objectives such as short wait times, high utilization, and selective prioritization. As the data-intensive applications that are common in cloud workloads are becoming increasingly common in HPC\textsuperscript{2}, a trend towards increased overlap between the paradigms can be observed.

This new diversity of workloads in supercomputing centers requires a thorough investigation of the right workload scheduling model required to allow CactoOpt to address the needs of diverse workloads within CACTOS and beyond. As a first step, we need to understand the evolution of the workload on the current systems in depth. We have thus collaborated with the Lawrence Berkeley National Lab to analyze the workloads at the National Energy Research Scientific Computing Center (NERSC), a supercomputing center that supports the broad scientific workload of the Department of Energy office of Science. Three systems were considered: Carver, Hopper, and Edison. They were selected due to differences in their hardware and timeline characteristics. Carver is a traditional high performance Linux cluster while on Edison and Hopper systems, commodity processors are packaged together on customized blades and connected with a high speed proprietary interconnect. The workload is composed of applications that belong to various scientific fields like Fusion, Chemistry, Material Science, Climate Research, Lattice Gauge Theory, Accelerator Physics, Astrophysics, Life Sciences, and Nuclear physics. The data includes 6 months and 1,023,503 jobs for Edison, 4.5 years and 4,326,870 jobs for Hopper and 4.5 years and 9,508,054 jobs for Carver.

**a) JOB CHARACTERIZATION**

Figure 1 shows the aggregated job characteristics of the three systems. We note that, a) significant share of the jobs run for 2h or less. b) Edison, Hopper: 68% of the jobs allocate 240 cores or less. Carver: 99% of the jobs allocate 1 node. c) Edison, Hopper: 25% of the jobs run over the requested time. Carver: 80% of the jobs run under 50% requested time. d) Carver receives significantly more job submissions per time unit. e) Jobs that wait less than 3h to be executed: Edison(70%), Hopper(60%), Carver(85%).

To get more insights on how some workloads can benefit from migrating to the cloud, the following table shows the percentage of jobs with no parallelism, and the percentage of jobs with lower than two hours running times. This is similar to MolPro, but many MolPro runs take considerably more time. While the LBNL workloads are mostly shorter jobs, with a significant percentage that in not parallelized.

<table>
<thead>
<tr>
<th>Job Distribution</th>
<th>Edison</th>
<th>Hopper</th>
<th>Carver</th>
</tr>
</thead>
<tbody>
<tr>
<td>%Jobs Wall Clock &lt; 2 h.</td>
<td>88%</td>
<td>86%</td>
<td>87%</td>
</tr>
<tr>
<td>%Jobs Width &lt; 240 Cores</td>
<td>62%</td>
<td>68%</td>
<td>99%</td>
</tr>
<tr>
<td>%Jobs Width = 1 Node</td>
<td>32%</td>
<td>33%</td>
<td>92%</td>
</tr>
</tbody>
</table>

**b) SCIENTIFIC WORKLOADS EVOLUTION**
The trend analysis studies the workload in sequential time periods. The size of the periods is calculated using a Fourier transform analysis on the number of tasks submitted per hour, which identifies cycles in the job submission and thus, the user behavior. This is shown in Figure 1. The most powerful cycles correspond to the periods of one day, one week, three months and six months, matching calendar work periods and allocation year. Each project has a number of core hours to be consumed during a year, divided in four allocation quarters in which the project has to consume (or forfeit) the corresponding allocated time. This increases the usage towards the end of the quarter increasing the overall queue time. These results indicate how quotas and costs can influence the way users use the system. Towards the end of each quarter, the job wait times in the queues increase which leads to delays in the results.

As we already saw, there are many jobs that are embarrassingly parallel running in these systems, these jobs can benefit from being offloaded to a cloud system as for many jobs the median runtime of the job is close to the media queuing time. Offloading to a cloud system would reduce the queuing time of the jobs to negligible time, and will thus speed up the time when the results are available.

In a traditional cloud system, it has been shown that many analytics and embarrassingly parallel jobs can improve their performance significantly making use of vertical scaling rather than horizontal scaling. To be able to improve the QoS of such applications, the recent version of CactoOpt supports vertical scaling besides horizontal scaling.

2. FAULT TOLERANCE OPTIMIZATION MODEL

Data centers achieve high reliability through the use of failure tolerant hardware, network equipment, and architectures or via adopting sophisticated management solutions such as replication and recovery techniques. Many of these techniques cope with failures of a single individual component such as a machine or a software component, rather than providing overall reliability for jobs or services. However, it is not generally valid to assume that machines fail independently and component failures are uncorrelated. Correlated failures such as failures due to power outages or network component failure are rare, but have significant effects on system reliability. Ignoring the impact of correlated failures can cause reliability to be overestimated by at least two orders of magnitude.

We have teamed up with researchers from Google to build a statistical model for job reliability in a cloud data center, in the presence of stochastic and correlated failures. The model quantifies the impacts of correlated failures on the overall reliability of a job comprising multiple identical tasks, which can be run in containers or virtual machines. The job reliability is defined as the probability that at least a minimum number of tasks will continue running throughout the job’s runtime. We do this to model batch jobs that need to have a certain number of workers to finish by a deadline, or long-running service jobs that need to have a minimum number of workers to meet some external load, such as a worst-case query rate.

---

Footnotes:


We specifically focus on correlated failures caused by power outages and failures of network components. In our model, power nodes and network components have different failure rates and their failures affect different sets of machines, known as failure domains. For example, the machines that are affected by a power outage will not necessarily be the ones that are affected by a network component failure.

Delivering job reliability cannot be considered independently from job scheduling because job reliability is highly dependent on the placement of the tasks over different failure domains (e.g., different racks or power failure domains). To achieve job reliability, scheduling and allocation decisions should take into account the probabilities of failure for different machines, racks, and other potential failure domains, along with the impact of failures in these domains on the job.

A job $J$ arrives at a data center and executes a group of identical tasks. This is a common application model for datacenters, where multiple tasks (containers or VMs) are needed to serve the job’s demand. Each task has an expected compute and memory demand of $C$ and $M$, respectively. The job starts at time $t_1$ and runs for time $T$. A job is successful if the probability that at least $K$ tasks will be running at all times during the $[t_1, t_1+T]$ time interval is greater than or equal to a threshold $S_{\text{min}}$. In addition to the required $K$ tasks, a number of extra tasks can be created to increase the expected probability of success. A set of tasks may fail (and stop running) simultaneously if they share a common source of potential failure such as a common power node or a common network device.

The tasks are deployed in a cluster of physical machines. Each machine $p$ has an available CPU and memory capacity of $c_p$ and $m_p$, respectively. Machines are connected by a set of network nodes $\mathbb{R}$ and the electrical power is supplied by a set of power nodes $W$. Failure of a power node causes failure on all the associated machines and the tasks deployed on those machines. Given a fixed power and network topology, for each job $J$ with desired number of running tasks $K$, we want to determine $n$, the required number of extra tasks, and $x$, a placement of $N = K + n$ tasks on the machines, so at least $K$ tasks are running at all times during the job runtime, with a certain probability threshold, $S_{\text{min}}$. The job reliability is defined as the probability of the job operating for a certain amount of time, with at least $K$ running tasks, at all times. In our model, each machine runs at most one task from each job. Tasks cannot be restarted and their failures are terminal.

Given a placement vector $x$ representing the distribution of tasks over different failure domains, and $S(x)$ as the calculated reliability of the job, the goal is to find $n^* = \min \{n \mid S(x) \geq S_{\text{min}}\}$ during the job’s runtime $T$, subject to the relevant capacity constraints.

\textbf{a) Network and Power Topology}

We model the dependencies among system components as a 2-level multi-rooted tree, where the leaves are the machines and the parents are power and network nodes. Each network component failure disconnects 60 to 120 machines (one or two racks), whereas a Power Distribution Unit (PDU) failure leads to the outage of 20 to 60 racks. We assume that the power and network topology may contain redundancy, where each machine can be connected to an extra network node or be supported by an extra power node. In such cases, a machine is functional as long as at least one power node and one network node are available to it.

We also define a network failure domain $R$ as the set of machines that share the same network components and are thus at risk of a concurrent failure. Similarly, we define a power domain $W$ as the set of machines that share the same power nodes.

We define $r_i$ as a random binary variable, where $r_i = 1$ if the network failure domain $i$ is available and running, and $r_i = 0$ otherwise. The network failure domain is available as long as one network component is functional. Similarly, we define the random binary variable $w_j$, where $w_j = 1$ if the power failure domain $j$ is available and $w_j = 0$ otherwise. The power failure domain is available as long as one power node is functional. In addition, we define the failure domain $F_i$, as the set of tasks deployed in a power failure domain that are also connected via a single network failure domain. The binary random variable $f_i = 1$ if at least one network node and one power node in $F_i$ are operational. Figure 1 shows a data center topology with respect to network and power failure domains, showing the redundancies in the system.

---

The network and power nodes break independently and are not repairable. The time to failure of each network node during a job runtime $T$ is exponentially distributed with a random failure rate of $0.000022 \leq \lambda \leq 0.000032$ per hour, given a Mean Time To Failur (MTTF) of 3.5 to 5 years. Similarly, the time to failure of the power nodes (i.e. the PDUs) during a job with runtime $T$ is assumed to be exponentially distributed over time with a failure rate of $\lambda = 0.4 \times 10^{-6}$ per hour. We also assume that all machines have an identical probability of hardware failure.

### b) The Reliability Model

The reliability model to model the impact of correlated failures on reliability, we specify the probability of each subset of tasks being unavailable. Let us assume that $x_i$ is the number of tasks running in a failure domain $F_i$, $x = [x_1, x_2, ..., x_i]$ is the placement vector of the tasks over the failure domains. The quantity $x_i$ also happens to be the number of tasks that fail when failure domain $F_i$ goes down. Moreover, $f = [f_1, f_2, ..., f_i]$ is a failure state vector showing the availability or failure of the failure domains and $L$ is number of failure domains. The number of running tasks can then be calculated as:

$$N(T, x) = \sum_{i=1}^{L} f_i x_i$$

Let,

$$P^R(r) = \prod_{\{i|t_i=1\}} A^R(T) \prod_{\{i|t_i=0\}} (1 - A^R(T))$$

$$P^W(w) = \prod_{\{j|w_j=1\}} A^W(T) \prod_{\{j|w_j=0\}} (1 - A^W(T))$$

where $P^R(r)$ and $P^W(w)$ are the probabilities of network failure domains and power failure domains for failure state vectors $r = [r_1, r_2, ..., r_i]$ and $w = [w_1, w_2, ..., w_j]$, respectively. In addition, $A^R(T)$ and $A^W(T)$ are the reliabilities of network failure domain $i$ and power failure domain $j$ for a job with $T$ runtime, respectively. The reliability of the job, $S(x) \geq P(N(f; x) \geq K)$, can be calculated as the sum of the probabilities of all possible combinations of failure events when the number of running tasks $N(f; x) \geq K$. This can be written as:

$$S(x) = \sum_{\{(r, w)|N(f; x) \geq K\}} P^R(r)P^W(w)$$

In this formulation, the component failures are independent of each other. However, each component failure can terminate all tasks in one or more failure domains.

### c) Approximating the Reliability Value

In practice, computing the exact reliability value is complex and computationally expensive. Therefore, we propose a method for approximating the reliability value. The reliability function $S(x)$ is a step function; it is assumed that a job's reliability increases in discrete steps as the number of extra tasks is increased. However, different types and combinations of failures do not necessarily reduce the reliability value to the same extent. A network node has a much greater failure probability than a power node, but fewer extra tasks are required to mitigate the impact of a network node failure.

We reformulate the last equation as a sum over the total number of failures in the system. To do this, for $i = 0, 1, ..., R$ and $j = 0, 1, ..., W$, we define:

$$S_{i,j}(x) = \sum_{\|r\|_1 = R - i}^{\|w\|_1 = W - j} P^R(r)P^W(w)$$

$$\{N(f; x) \geq K\}$$

In this equation, $S_{0,0}(x)$ is the reliability of the job if none of the network or power failure domains fail, and $S_{i,j}(x)$, $1 \leq i \leq R$ and $1 \leq j \leq W$ is the probability that the job has $N(f; x) \geq K$ tasks, given $i$ network failures and $j$ power failures. By combining the last two definitions, we get the following expression for the reliability,
The probability of failure of multiple components, $S_{ij}(x)$, depends on the failure probabilities of the individual components, each of which is small. Considering the probability of failures of the components in our model, we can argue:

$$S_{10}(x) \gg S_{10}(x) \gg S_{20}(x) \gg S_{30}(x) \gg S_{0}(x)$$

Therefore, as $i$ and $j$ increase, the improvement in system reliability, $S_{ij}(x)$, becomes progressively smaller. There is thus an optimal number of extra tasks; adding further tasks above this threshold would not contribute substantially to the system’s reliability. In other words, as the probability of a high number of failures during a job’s lifetime decreases, so too does the need to plan and assign extra tasks to achieve reliability.

The fact that the reliability function increases in discrete steps is useful when approximating the reliability value and estimating the number of extra tasks required to achieve a given reliability. Each step in the reliability function corresponds to a failure arrangement, $(r,w)$, which specifies the type and number of failures that the existing arrangement is sufficient to cover. A desired reliability can then be achieved by providing the minimum redundancy required to cover the corresponding failure arrangement.

The approximation becomes essential, because of the computational expense of calculating $S(x)$, which necessitates calculation of every possible combination of failures that satisfy $N(f(x) \geq K)$. To reduce the computational complexity, we approximate the sum on the right hand side of the last equation by discarding the terms $S_{ij}(x)$ that correspond to failures of components $(i,j)$ whose probability of failure is negligible with respect to $S_{\text{min}}$. This enables us to obtain approximate reliability values cheap, and also helps us estimate the number of extra tasks required to prevent failure events that have non-negligible effects on reliability relative to $S_{\text{min}}$.

**d) Effect on Optimization**

To schedule a job in a way that achieves a given reliability, we must approximate the minimum necessary number of extra tasks and identify a placement that satisfy the reliability constraint $S(x) \geq S_{\text{min}}$. Our approximation algorithm is based on the discussion in the previous section. We aim to determine which failures of $f(i,j)$ must be compensated for to achieve the target reliability $S_{\text{min}}$. Having identified this set of essential failures, we then provide the minimum level of redundancy necessary to cover them.

To identify the failures for which it is necessary to provide redundancy, we implement a decision tree. Each node in the tree corresponds to a failure of $(i,j)$ network and power failure domains. For each node of the tree, we estimate the number of extra tasks required to compensate for the corresponding failures, $n$, and approximate the job reliability for the current arrangement. The algorithm starts at the root of the tree $(0,0)$, and initially expands along the power branch $(0,1)$. This is done because providing redundancy for power outages with appropriate task placement also automatically protects against some network failures. If the target reliability, $S_{\text{min}}$, cannot be achieved by covering for one power failure, no amount of additional redundancy with respect to network failure would be sufficient to compensate for this deficiency, so it is necessary to expand further along the power branch. However, if providing redundancy for one power failure domain’s failure results in $S(x) \geq S_{\text{min}}$, there is a chance of obtaining the desired reliability with fewer extra tasks by covering for just some additional network domain failures. If this is the case, we expand along the network branch and iteratively increase the number of possible component failures, $(i,j)$, adding the necessary redundancy to cover these failures at each step. In each expansion, we re-compute the new $x$ for new $N = K + n$ and its associated $S(x)$. We stop at the node that satisfies the target reliability $S_{\text{min}}$.

The $n$ values of the last two nodes are the lower and upper bound estimates of the approximate minimum number of extra tasks required to achieve $S(x) \geq S_{\text{min}}$. Having determined this interval, we can easily approximate the minimum by performing a bisection search over the $n$ values that satisfy $S(x) \geq S_{\text{min}}$. The number of iterations is limited and small because the bound is usually limited and small. This algorithm is outlined graphically in the following figure and more precisely in Algorithm 1.

The proposed algorithm is not easy to test on normal testbeds (large or small) in a real setting, as a) it requires a full datacenter with many PDUs and network segments and, b) it requires injecting such failures to test the algorithm. We provide analytical proofs of the algorithm’s validity for a cluster with different failure sources and jobs with different target reliabilities. In D7.4.2 we have also noted that the failure rates we saw...
during the project’s life time were too low to really work on that aspect in our testbed. Nevertheless, we see this algorithm as a contribution towards enabling CactoOpt to handle large datacenters.

While not currently part of the CACTOS toolkit, in very large scale datacenters, the tasks are usually replicated for fault tolerance, increasing the overall reliability and availability of the datacenter applications. To estimate the number of extra tasks \( n \) needed to add extra reliability, let us first assume that the desired reliability \( S_{\text{min}} \) can be guaranteed by covering a single network failure domain:

**Lemma 1.** To provide full redundancy for one network domain failure, the required number of extra tasks is

\[
 n = \left\lceil \frac{K}{R-1} \right\rceil.
\]

**Proof.** Let \( N \) be the total number of deployed tasks for the job. To ensure that at least \( K \) tasks are running if any single network failure domain fails, at least \( K \) tasks must run on each collection of \( R-1 \) network failure domains. By the pigeon hole principle\(^6\), at least one of these \( R-1 \) network failure domains has \( n \) tasks. To ensure redundancy, the remaining \( R-1 \) network failure domains must have at least \( K \) tasks, so we need \( K+n \) tasks to ensure redundancy. Furthermore, we need to prove that this setting is realizable. In other words, we should be able to place \( K+n \) tasks over the \( R \) network failure domains such that no domain has more than \( n \) tasks. One straightforward way of doing this is to put \( n \) tasks on each network failure domain, resulting in

\[
 R \left\lceil \frac{K}{R-1} \right\rceil \geq K + n
\]

tasks in total, and then remove any \((Rn)-K-n\) tasks.

**Lemma 2.** To provide full redundancy for \( i > 0 \) network domain failures, the number of required extra tasks \( n_i \) is equal to

\[
 i \left\lceil \frac{K}{R-i} \right\rceil.
\]

---

Proof. Lemma 2 is an extension of the statement in Lemma 1. For a job to survive $i \geq 1$ network domain failures, we need for any combination of remaining $R - i$ domains to have at least $K$ tasks. The simplest way is to assume that we have the same number of tasks $n$ in all $R$ domains. Then $Rn \geq K$ or $n \geq K/(R - i)$, that is when we can take the smallest number $n = \lfloor K/(R - i) \rfloor$ where $m = \lfloor a \rfloor$ is the smallest integer number such that $m \geq a$. Now we obtain

$$n = \left\lfloor \frac{K}{R - i} \right\rfloor \geq \frac{K}{R - i} \iff Rn \geq K + in.$$ 

Therefore, we get the number of extra tasks

$$n_r = in = i \left\lfloor \frac{K}{R - i} \right\rfloor.$$ 

Using a similar argument, we can conclude that the number of extra tasks required to provide full redundancy in the event of $j > 0$ power failures is

$$n_w = \left\lfloor \frac{K}{W - j} \right\rfloor.$$ 

The only difference is that deploying $n_w \gg n_r$ tasks automatically covers for the failure of $R^*$ network failure domains due to the deployment of a greater number of spare tasks. Using the equation in Lemma 2, we can find $R^*$, the number of network domain failures covered by deploying $n_w$ extra tasks:

$$n_w = \left\lfloor \frac{K}{R - R^*} \right\rfloor \rightarrow R^* = \left\lfloor \frac{n_w}{K + n_w} \times R \right\rfloor.$$ 

Therefore, when estimating the increase in reliability it is necessary to consider the coverage of both the power failures and $R^*$ network failure domain failures.

To identify a suitable placement given the failure of $(1, j)$ components, we must ensure that at least $K$ jobs are running on each combination of failure domains. One intuitive approach is to evenly assign $n$ tasks to each domain. Assuming that placement is done over network failure domains, this results in at most $Rn \geq K + n$ tasks in total given the capacity constraints. Next, we can remove $(Rn) - K - n$ tasks from the assigned tasks. Since $n$ is derived independently from the failure probabilities of each failure domain, a lower $n$ can be obtained by removing the tasks from the failure domains with highest probability of failure. We continue reducing the number of tasks from the least reliable failure domains as long as the reliability condition $S(x) \geq S_{\text{min}}$ holds. If the placement is to be done over power failure domains, we first assign tasks to each power failure domain and then distribute the tasks over the network failure domains within the power failure domain to cover the failures of network failure domains.
3. POWER-PERFORMANCE MODELS AND TRADEOFFS

There are two main approaches of physical server resource management that have been proposed in the literature to improve the energy efficiency of data center servers: server consolidation and server throttling. Server consolidation is one of CactoOpt supported optimization goals. It reduces the number of physical servers needed to host a workload through collocation of applications thus enabling some servers to be powered down while the rest run at the high utilization level. This is more energy efficient than operating at the lower levels. However, the server consolidation in a data center with dynamic workloads involves migrations of virtual machines between physical servers in order to reconsolidate workload on a minimal subset of physical servers. Virtual machine migrations are costly as they negatively impact the performance of migrated applications, and increase the utilization of physical resources, especially the network, during the migration. Moreover, in some cases powering down physical servers is not possible because hosted applications may need some of the resources
on the lightly loaded machines, e.g., storage in a networked filesystem. Therefore, workload consolidation cannot always be used to effectively improve the energy efficiency of a data center. Server throttling techniques include Dynamic Voltage and Frequency Scaling (DVFS) and server power capping. This approaches are orthogonal to server consolidation and to some extent can be used in conjunction with it.

In this chapter we focus on the power-performance tradeoffs involving three main actuators used to optimize datacenter servers, namely DVFS, vertical, and horizontal scaling. We consider two orthogonal dimensions when studying the power-performance tradeoffs of the four actuators, namely: a) the effect of hardware architectures, and b) different workload intensities and distributions.

**a) Description**

We ran our experiments on two physical server architectures, namely:

1. HP ProLiant DL165G7 servers equipped with 32 CPU cores (AMD Opteron 6272, 2:1 GHz), 56 GB of RAM, and 4x500 GB SATA disks arranged in RAID 1+0. The CPU has two sockets, each socket has two Non-Uniform Memory Access (NUMA) nodes, and each NUMA node has 8 cores. Moreover, the cores are arranged in pairs and core pairs share a Floating Point Unit and L2 cache. CPU cores on a single die share L3 cache. The idle power consumption of the machine is 120W. The cores can operate at five frequencies: 1.4, 1.5, 1.7, 1.9, and 2.1 GHz.

2. Dell R530 servers with 12 CPU cores (Intel Xeon CPU E5-2620 v3, 2:4 GHz) and 64 GB of RAM. The storage consists of 3 Intel DC3610 SSDs and 3x Dell 7200 rpm 4 nearline-SAS HDD, both arranged in a hardware RAID 5. There are two CPU sockets, each socket has 6 cores (hyperthreading is disabled for our experiments). The cores can operate at frequencies between 1:2 and 3:2 GHz. However, frequencies above 2:4 GHz are turbo frequencies, which means that they can be achieved only on a subset of CPU cores, if that does not cause overheating or power over-consumption. The idle power consumption of the machine is 78W.

We monitor the power consumption of physical servers using HP Intelligent Modular PDUs which provide per-power-socket power usage over Simple Network Management protocol. The measurements are done every 0.5 s with accuracy within 1W.

In our experiments, we used two different applications with different profiles.

1. We use stress to simulate a CPU bounded application. Stress is a workload generator for POSIX systems that allows to stress CPU cores.
2. MediaWiki, which is a custom-made, free and open-source wiki software platform written in PHP and JavaScript. The installation run on Ubuntu Linux. The frontend is an Apache webserver, enhanced with a MemCached installation. The backend is a MySQL database replicating the German Wikipedia. In all experiments with multiple virtual machines, we use HAProxy running on a separate physical server for load balancing. The VMs run on KVM.

For generating the workload, we have implemented a workload generator using Python that is able to produce different workload profiles. The main workload profiles used in our experiments are:

1. **Constant**: the number of requests generated per second does not change over the time of experiment, and the experiment lasts for r repetitions of request generation.
2. **Step**: the number of requests generated per second grows from 0 to a predefined value m by step of s requests and each step is repeated r times. For example, from 0 to m=140 requests per second with a step of s=5 requests and r=2 request generation per step, the number of requests generated per each second will look as follows: 0, 0, 5, 5, 10, 10, ..., 135, 135.
The request generation process has three modes:

- **Concurrent**: a client generates a batch of \( n \) requests in parallel at the beginning of each second,
- **Evenly spread**: a client generates a single request every \( 1/n \) of a second.
- **Poissonian**: the time between requests generation is determined by a Poisson process.

The first two generation modes are extreme cases of all possible distributions of \( n \) requests over a second. The distribution of any real application lies somewhere in between these two. Poisson process is commonly used to model the request arrival time. Three request generation modes at the workload generator side translate to three types of request arrival pattern at the server side.

### b) DVFS

Scaling the frequency and voltage of a physical CPU is used to change the performance capabilities and power consumption of a physical server. The power consumption of a CPU depends on both the frequency and the voltage (with voltage having a bigger impact). Each CPU frequency is coupled with an appropriate voltage level and a change of CPU frequency causes a simultaneous change of the voltage. For multi-core processors with a single voltage regulator, such as our AMD processor, the voltage is associated with the highest CPU frequency in use by any core at given time.

![Power consumption graph](image)

**Figure 3** The difference in power consumption between various CPU frequencies grows with the number of fully utilized cores.

First, we analyze an extreme case, when an application uses only CPU resource (using stress application) to determine the maximal potential benefit of using DVFS in terms of power savings. We focus solely on the power consumption of a physical server.

Figure 3 shows the average values of the power consumption for the same load calculated from more than a thousand measurements taken over one hour for each frequency setting on the AMD processors. We test five CPU utilization levels: all cores idle, 4 cores fully utilized and the rest idle, 8 cores fully utilized and the rest idle, 16 cores fully utilized and the rest idle, and 32 cores fully utilized; at the five available CPU Frequencies. An idle physical server running with the lowest frequency (1.4 GHz) consumes approximately 95% of the energy consumed at the highest frequency (2.1 GHz). For a fully utilized physical server the difference is bigger, running a physical server at the frequency of 1.4 GHz saves around 18% of energy when compared to a physical server operating at the frequency of 2.1 GHz.
To test the effect of DVFS on request-response application, we perform a set of experiments quantifying the influence of workload characteristics on the performance of a MediaWiki application hosted on a physical server that operates on various CPU frequencies with the workloads described above.

Figure 4 Influence of DVFS on the response time (RT) and power consumption for AMD servers. Dashed horizontal lines depict the observed thresholds of system saturation. The impact of DVFS on an underloaded server (up to 40 requests) depends on the request arrival pattern. The average and tail response time of an overloaded application stabilises because of the limit on not processed requests.
Figures 4 and 5 show the influence of workload intensity (x-axis), CPU frequency (data series), and request arrival pattern (each column) on the average response time (the first row), tail response time (the seconds row), maximal throughput (the intersect of data series and the dashed horizontal line), and the power consumption (the third row) for AMD and Intel servers.

Let us first consider the case of a physical server with AMD processor and a workload with evenly spread requests. For the AMD processors, DVFS has an influence on the performance of the application, changing the size of workload that can be handled in underloaded state from 40 requests for 1.4 GHz to 60 requests for 2.1 GHz. The difference in power consumption, is much lower for an underloaded application (around 10W) compared to the difference when the application is overloaded (around 40W).

For the Intel processors, one can observe that operating at the lowest CPU frequency (1.2 GHz) causes a significant performance degradation when the application is overloaded, increase in the response time of approximately 3 seconds while saving about 10W. On the other hand, using turbo frequency (3.2 GHz) reduces the response time about 1 second at the cost of approximately 20W.

For the AMD machines, when the requests are generated concurrently, the power consumption of an underloaded physical server (with a workload up to 40 concurrent requests for 1.4 GHz and 60 requests for 2.1 GHz) is not significantly affected by the CPU frequency. However, lowering the CPU frequency limits the power consumption when the server is saturated.

Seeing the tradeoffs using a workload with evenly spread requests, we study if changing the request arrival pattern has an effect on the power-performance tradeoffs. When the requests are generated concurrently, the power consumption of an underloaded AMD physical server (with a workload up to 40 concurrent requests for 1.4
GHz and 60 requests for 2.1 GHz) is not significantly affected by the CPU frequency. However, lowering the CPU frequency limits the power consumption when the server is saturated. Therefore, we conclude that DVFS is not able to reduce the power consumption of an underloaded physical server when the requests are generated concurrently. However, for an overloaded application (workload higher that 40 requests) the difference in the power consumption becomes significant.

The request arrival pattern has a significant impact on the power consumption of an underloaded application, since it influences the number and duration of intervals that CPU cores spend in an idle state. When the requests are spread evenly along each second of the workload, the server with Intel processor consumes approximately 20W more for a workload of 5 and 10 requests per second, comparing to the concurrent requests. In an extreme case of concurrently generated requests and an underloaded application, decreasing the CPU frequency even causes an increase in power consumption. However, when the workload intensity increases, the difference due to the arrival pattern becomes less important, and finally, when the application becomes overloaded it does not have any measurable impact.

c) **Horizontal vs Vertical Scaling consolidation effects**

Changing the number of virtual machines hosting application instances (horizontal scaling) or the amount of resources allocated to a virtual machine (vertical scaling) are two optimization capabilities that have been developed and added within the third year to CactoOpt. Since both horizontal and vertical scaling change the amount of physical resources used to serve the application workload, they have an indirect influence on the power consumption of servers hosting the virtual machines. Their performance is highly dependent also on how the VMs are consolidated.

To investigate the power-performance tradeoffs of vertical and horizontal scaling, we perform a set of experiments using the MediaWiki application with Memcached. We evaluate the costs and benefits of scaling the application vertically (with respect to the number of CPU cores and size of RAM allocated to a single virtual machine) and horizontally (with respect to the number of virtual machines) when the VMs are consolidated to capture the interference effects. In the horizontal scaling experiments we use virtual machines with 8 CPU cores and 10 GB of RAM. We scale the number of virtual machines from 1 to 4 instances. All instances are hosted on a single server to evaluate its power-performance tradeoffs. We run all experiments on the AMD machines.
Figure 6 shows the evaluation results for the relation between the number of virtual machines and the throughput of the application exposed to a step workload. Scaling out (increasing the number of virtual machine instances) extends the maximum number of requests that the application is able to handle before becoming overloaded. However, after reaching some number of virtual machines (3 virtual machines in our case, with 75% of available CPU cores allocated) further scaling out is not beneficial, showing a non-linear relationship between the response time and allocated capacity. With a threshold on the average response time on 1 second an application consisting of a single virtual machine can handle up to 35 requests, two virtual machines allow to handle a workload up to 65 requests, and an application consisting of three or four virtual machines can serve up to 90 requests. Possible reasons for the lack of improvement when adding the fourth virtual machine are saturation of some resources, inter-vm switching, or background jobs (e.g. operating system) that reduce the pool of available resources.

We also investigate the effects of scaling out on the average and tail response times. Horizontal scaling does not affect the response time when the application is underloaded. When one or few instances can handle the workload, adding another instance does not improve the response time significantly. For an overloaded application, increasing the number of virtual machines reduces the average response time, however that reduction is not proportional to the amount of resources added. The average response time of an overloaded application stabilizes at a higher level when fewer virtual machines are used, because the application is overloaded to a higher extent. For example, for a system with a single virtual machine, while the application is able to serve only 35 requests per second before overloading, there are up to 200 requests in the system (including delayed ones). Changing the limit on the maximum number of requests present in the system has a significant impact on the average time of overloaded application. For example, the average response time of a system with two virtual machine exposed to a constant workload of 100 requests per second, varies from 1.6 s for a limit of 100 requests, through 3.2 s for 200 requests, to 4.7 s for 300 requests. Moreover, scaling the system...
horizontally while using the round robin load balancing policy (a default one in HAProxy) does not improve the tail response time when the application is overloaded.

Now looking at the power consumption, Scaling in (decreasing the number of virtual machines) does not reduce the power consumption of a physical server when the application is underloaded. The limited number of virtual machines influences only the maximal power consumption, when the application is overloaded.

Finally, CPU pinning has an influence on the power-performance tradeoffs for horizontal scaling. Pinning virtual CPUs to the first \( n \) physical CPU cores negatively impacts the performance of an application with one and two virtual machines. For these configurations, the system is able to serve a lower number of requests under the same thresholds compared to the unpinned configurations. On the other hand, pinning improves the performance of an application that uses all the CPU cores (four virtual machines in our case). In this configuration the system can serve up to 105 requests under the thresholds (compared to 90 requests, when virtual CPUs were not pinned). CPU pinning decreases the power consumption of the system with one, two, and three virtual machines, while slightly increasing the power consumption of a system with four virtual machines.

To understand why the tail response time is not improved by horizontal scaling we have performed additional experiments. We have observed that when we use the default strategy for load balancing in HAProxy - round robin - some virtual machines got overloaded more than the other, and their performance was much worse. That does not affect the average response time, since the other virtual machines still perform well, and the vast majority of requests is processed in short time. However, the overload of even a single virtual machine has a big impact on the tail response time, because it affects enough requests to significantly increase the 95 percentile of response time. Therefore, we repeat the experiment for horizontal scaling with a different load balancing policy provided by HAProxy - least connections - that directs new requests to the server with the lowest number of connections.

Figure 7 shows the influence of workload intensity (x/axis), CPU pinning (columns), and a number of virtual machines (data series), on the average response time (the first row), tail response time (the second row), maximal throughput (the intersect of data series and the dashed horizontal line), and the power consumption (the third row). The change of the load balancing strategy results in a lower tail response time for configurations with two, three, and four virtual machines compared to the configurations with the round robin load balancing strategy in the previous figure. However, that improvement comes at a cost of a degraded average response time of an application with two, three, and four virtual machines. The power consumption is not affected significantly by the change of the load balancer strategy. Also the influence of CPU pinning is very similar to what we have observed for the configurations with round robin load balancer strategy. We conclude that the load balancing strategy can drastically change the characteristic of the tail response time of a horizontally scaled application. Physical servers are complex heterogeneous systems, and the performance of virtual machines hosted on a single server may differ due to saturation of some shared components. Therefore, a load balancing strategy that constantly considers the performance of virtual machines and avoid overloads are needed to keep the tail response time low. These can be done by monitoring the number of pending requests (a queue length) or the current performance of virtual machine (the response time).
Turning our focus to vertical scaling (changing the amount of physical resources allocated to a virtual machine), in the vertical scaling experiments we use a single virtual machine and change both the number of the allocated CPU cores and the size of RAM. We evaluate four configurations: 8 CPU cores with 10 GB of RAM, 16 cores with 20 GB, 24 cores with 30 GB, and 32 cores with 40 GB.

First, we analyze the influence of the vertical scaling on the throughput. Clearly, the smallest virtual machine (with 8 CPU cores and 10 GB of RAM) has worse performance than the bigger virtual machines. A virtual machine with 8 cores is able to handle close to 30 concurrent requests with the average response time under 1 second as shown in Figure 6. By scaling up the virtual machine by 8 cores and 10 GB of RAM (up to 16 cores and 20 GB of RAM in total) one extends the ability of the application to handle the workload of up to 65 requests with the average response time under 1 second. Adding another 8 cores and 10 GB of RAM (a virtual machine with 24 cores and 30 GB of RAM) results in an ability to serve up to 80 requests while not overloaded. Finally, a virtual machine with 32 cores and 40 GB of RAM is able to handle up to 95 requests with an average response time under 1 second.
Second, we investigate how vertical scaling affects the response time. Similarly, to horizontal scaling, adding more resources reduces the average response time as shown in Figure 6 and the tail response time of an overloaded application. While scaling up from 8 to 16 CPU cores reduces the response time by around 50%, further scaling up does not give so significant improvements in the application performance. Both the average and tail response time stabilize at the similar level for virtual machines with 16 CPU cores and more. The power consumption, shown in Figure 6, differs significantly only for a virtual machine with 8 cores, when the application is overloaded (for a workload over 40 requests).

For vertical scaling, CPU pinning improves the performance of very large virtual machines utilising most of the CPU cores of the host physical server (24 and 32 cores in our experiments, as shown in Figure 6). The largest virtual machine is able to serve up to 105 requests with an average response time under 1 second comparing to 95 requests for a setting without CPU pinning. Also the average response time of an overloaded application decreases by approximately 1 second. CPU pinning has a significant influence on the power consumption when an application is scaled vertically. First, CPU pinning makes the power consumption of virtual machines with different sizes more differentiable. While for not pinned virtual machines the difference in power consumption among virtual machines with 16 and more CPU cores allocated is smaller than 25W, the difference between the 16 cores virtual machines and the 32 cores virtual machine is around 60W when virtual CPUs are pinned. CPU pinning decreases the power consumption of small and medium virtual machines that utilise a half or less of CPU cores of the host physical server (8 and 16 cores in our case), because the remaining part of CPU can enter idle states. Third, the power consumption of the virtual machine with 32 cores allocated is higher when virtual CPUs are pinned.

d) Discussion

Based on the obtained results, we prepared a set of recommendations for using DVFS, CPU pinning, and horizontal and vertical scaling taking into account the power-performance tradeoffs in CactoOpt. These recommendations are presented in the following table. We list various conditions that describe the initial state of the system, together with recommended actions and their influence on the power consumption (PC) and response time (RT).
The experimental results have shown that DVFS, CPU pinning, and virtual machine scaling have a significant influence on the power consumption only when the physical server is highly utilized. There are two main reasons for that. First, at the low utilization levels the total power consumption of a physical server is highly influenced by the idle power consumption part. The above-mentioned techniques affect only the dynamic part of power consumption and therefore do not reduce significantly the total power consumption. Second, when the system is not highly utilized, the CPU spends a lot of time in the idle states, when DVFS, CPU pinning and virtual machine scaling techniques simply do not influence the power consumption.
III. NEW OPTIMIZATION ALGORITHMS

1. AUTOSCALING

a) HORIZONTAL SCALING ALGORITHMS

Five state of the art new horizontal autoscaling algorithms have been added to CactoOpt.

REACT

This algorithm is based on the algorithm designed by Chieu et al. It is a dynamic scaling algorithm for automated provisioning of VM resources based on the number of concurrent users, the number of active connections, the number of requests per second, and the average response time per request. The algorithm first determines the current application instances with active sessions above or below a given utilization. If the number of overloaded instances is greater than a predefined threshold, new web application instances are provisioned, started, and then added to the front-end load-balancer.

In our implementation, we base the decisions on the number of requests served per second and the request arrival rates. When more than 2 machines in the system become underutilized, the scale-down mechanism is triggered. Our implementation guarantees that there are at least 2 servers extra running to make sure that there is a buffer of resources enough to handle a spike occurring early on before new machines can be spawned. The main reason we include this algorithm CactoOpt is that this algorithm is the baseline algorithm in our opinion since it is one of the simplest possible workload predictors.

ADAPT

This is an autonomous elasticity controller that changes the number of VMs allocated to a service based on both monitored load changes and predictions of future load. We refer to this technique as Adapt. The predictions are based on the rate of change of the request arrival rate, i.e., the slope of the workload, and aims at detecting the envelope of the workload. The designed controller Adapts to sudden load changes and prevents premature release of resources, reducing oscillations in the resource provisioning. Adapt tries to improve the performance in terms of number of delayed requests, and the average number of queued requests, at the cost of some resource over-provisioning.

HIST

This algorithm is based on the algorithms proposed by Urgaonkar et al. They propose a provisioning technique for multi-tier Internet applications. The algorithm adopts a queuing model to determine how many resources to allocate in each tier of the application. A predictive technique based on building Histograms of historical request arrival rates is used to determine the amount of resources to provision at an hourly time scale. Reactive provisioning is used to correct errors in the long-term predictions or to react to unanticipated flash crowds. We refer to this technique as Hist. In our implementation, we use the 95th percentile historical request arrival rate data for the past a few weeks for a given hour of the day as a prediction for the expected load on that hour the next day. We couple this with a reactive predictor to correct errors in this expected value coupled with the average error in our predictions for the past a few hours.

REG
This implemented algorithm is based on a proposed a regression based autoscaler (hereafter called Reg) in the literature. The autoscaler has a reactive component for scale-up decisions and a predictive component for scaledown decisions. When the capacity is less than the load, a scale up decision is taken and new VMs are added to the service in a way similar to React. For scale down, the predictive component uses second order regression to predict future load. The regression model is recomputed using the complete history of the workload when a new measurement is available. If current load is less than the provisioned capacity, a scale down decision is taken using the regression model. This autoscaler was performing badly in our experiments due to two factors; first, building a regression model for the full history of measurements for every new monitoring data point is a time consuming task. Second, distant past history becomes less relevant as time proceeds. After contacting the authors, we have modified the algorithm such that the regression model is evaluated for only the past 60 monitoring data points.

CONPAAS
ConPaaS, proposed by Fernandez et al. scales a web application in response to changes in throughput at fixed intervals of 10 minutes. The predictor forecasts the future service demand using standard timeseries analysis techniques, e.g., Linear Regression, Auto Regressive Moving Average (ARMA), etc. The code for this autoscaler is open-source. As this implementation is open sourced, we use the authors’ source code.

b) Vertical Scaling
Vertical elasticity involves adding or removing resources such as CPU cores and memory to or from a VM. It requires little support from the application, which in essence only needs to be multi-threaded; the elasticity is primarily supported by the hypervisor and the guest operating system’s kernel. Horizontal elasticity is generally course grained. For example, a whole core may be exclusively allocated to a VM for a comparatively long period such as an hour. Conversely, vertical elasticity is fine-grained: individual fractions of a core may be allocated to a VM for as little as a few seconds. Vertical elasticity could thus be a key enabling technology for resource-as-a-service clouds, infrastructures that lease resources at CPU-cycle and second granularities. Such infrastructures would benefit cloud users by allowing them to pay only for the resources they actually use, and cloud providers by allowing them to use their resources more efficiently and serve more users. In fact, the technological cornerstone of the resource-as-a-service concept has arguably already been laid by the development of lightweight virtualization frameworks such as LXC and commercial offerings such as dotCloud.

Performance Models for Vertical Scaling
Performance models need to satisfy several constraints. First, due to the heterogeneity of hosted applications, the performance models need to be as generic as possible and should require no knowledge of application internals. Second, they should predict average behavior and ignore sporadic noise observed in the past. Such noise may be caused, for example, by variation in data retrieval: some data may be cached in memory, while other information may have to be fetched from disk. Third, these performance models should quickly adjust to variations in workload and capacity requirements. For example, an increase in the number of users or a change in request distribution may necessitate model parameter refitting.

Ideally, when given an input Key Performance Indicator (KPI) value, a performance model should return the exact capacity that must be allocated to an application to achieve that KPI value. Since this is difficult to achieve in practice, a performance model should at least drive capacity allocations towards the correct value. That is to say,

---

the model parameters used to estimate capacity requirements should be updated periodically so that the application eventually achieves the desired performance.

The load of a cloud application can be of three types: open, closed or partly-open\textsuperscript{11}. In an open-system model, which is typically described using a Poisson process, requests are issued with an exponentially-random inter-arrival time characterized by a rate parameter, without waiting for requests to actually complete. Conversely, in a closed system model, a number of users access the application, each executing the following loop: issue a request, wait for the request to complete, “think” for a random time interval, repeat. The resulting average request inter-arrival time is the sum of the application’s average think- and response times; it therefore depends on the application’s performance. A partly-open system model is an intermediate between the open and closed models: users arrive according to a Poisson process and leave after some time, but behave in a closed fashion while in the system. As with the closed model, the inter-arrival time depends on the performance of the evaluated system.

When developing our performance models, we start by assuming an open system. The response time in such a scenario increases more rapidly as the system approaches saturation. Therefore, vertically elastic capacity allocation must be done very carefully.

End-users of interactive applications are sensitive to response times; several studies have shown that increased response times reduce revenue\textsuperscript{12}. It is therefore desirable to maintain a given target response time for each application. Unfortunately, it is difficult to model response times due to their non-linear relationship with capacity. We use two different response time models: the queue length model, and a novel inverse model.

1) **Queue Length Model**: Starting from Little’s Law, the relationship between the average response time $R$ and capacity $c$ for an application can be represented as:

$$q = \lambda \cdot R,$$

where $q$ is the average queue length, i.e. the number of requests that have entered the application but have yet to exit, and $\lambda$ is the arrival rate. Besides, the mean response time given by the M/M/1 queuing model is:

$$R = 1/(\mu - \lambda),$$

where $\mu$ is the application’s average service rate. The relationship between capacity and response time can be modeled as $\mu = c/\alpha$, where $\alpha$ is a model parameter. Using simple substitution, one obtains the following expression for the mean response time:

$$R = \alpha(q+1)/c,$$

where $\alpha$ is a model parameter and $q$ is the number of requests waiting to be serviced. The parameter $\alpha$ can be estimated online from past measurements of the average response time, average queue length and capacity, thus compensating dynamically for small non-linearities in the real system. However, to reduce the impact of measurement noise, we decided to use a Recursive Least Square (RLS) filter. In essence, such a filter uses past estimates of $\alpha$ and the current ratio $(Rc)/(q+1)$ to output a new value that minimizes the least-squares error. A forgetting factor is used to trade the influence of old values for up-to-date measurements.

2) **Inverse model**: We model the inverse relationship between an application’s mean response time $R$ and the capacity allocated to it as:

$$R = \theta/c,$$

where $\theta$ is a model parameter. As in the queue length model, the parameter $\theta$ can be estimated using past measurements of capacity and average response time using the above equation. As before, to reduce the influence of measurement noise, we use an RLS filter. Note that the inverse model needs less information from the application than the queue length. In our experiments, we recompute the capacity allocated to the application

\textsuperscript{12} F. Fui-Hoon Nah. A study on tolerable waiting time: how long are web users willing to wait? Behaviour and Information Technology, 23(3), 2004.
periodically, with a control interval of 5 seconds. This is short enough to make the system reactive and long enough to observe the effects of the new capacity allocation on the application’s performance and the overhead of reallocation is negligible. Besides, we use a forgetting factor of 0.2, which was determined experimentally, for both models.

c) **Integration with CactoOpt**

The autoscaling algorithms run inside autoscaling services that are implemented as REST services independent from the cyclic optimiser. The interfaces that cyclic optimiser uses to communicate with the autoscaling services are described below.

The horizontal autoscaler service takes the following arguments:

- autoscaler algorithm – specifies which horizontal scaling algorithm shall be used (all algorithms are described in Section 3.3),
- server speed – the number of requests that each virtual machine hosting an application instance can handle,
- capacity – the number of virtual machines hosting an application instance that are running currently,
- load request – the total number of incoming requests measured at the load balancer,
- application identifier – specifying the application (and the application layer if applicable),
- current time (for simulation only) – to pass the simulation time for algorithms that are time dependent.

The horizontal autoscaler is stateful and it stores time series of load requests for each application for all calls that were made.

The response of the horizontal autoscaler service is a single value that indicates the suggested change of the number of application instances (virtual machines):

- a positive value means that n new virtual machines shall be added,
- a negative value means that n existing virtual machines shall be stopped,
- 0 means that the service suggest to keep the number of virtual machines as it is.

The vertical autoscaler service takes as an argument a list of servers, and for each server:

- the total number of CPU cores,
- a list of hosted virtual machines, and for each virtual machine:
  - the maximal number of CPU cores,
  - a target response time,
  - a measured response time.

The response of the vertical autoscaler service is a list of servers, and for each server it specifies:

- a list of hosted virtual machines, and for each virtual machine:
  - the suggested number of CPU cores assigned.

d) **Evaluation of the Horizontal Scaling Algorithms**

We have thoroughly evaluated our implementations of the horizontal scaling algorithms in both simulations, and in real deployments. In the interest of space, for a thorough overview of the evaluation results we point the reader
towards two recent publications in ACM Transactions on Modeling and Performance Evaluation of Computing Systems\textsuperscript{13} and in the International Conference on Performance Engineering\textsuperscript{14}.

\textbf{e) Evaluation of the vertical scaling models}

Evaluations were performed on a single server similar to the AMD servers used in the power-performance tradeoff modeling described above. To emulate a typical cloud environment and easily enable vertical elasticity, we used the Xen hypervisor. Each tested application was deployed with all of its components such as web servers and database servers inside its own VM as is commonly done in practice, e.g. by using a LAMP stack [1]. Since we were primarily interested in evaluating CPU allocation strategies, we configured each VM with 6 GB of memory, enough to avoid disk activity. To test that our algorithms are applicable to a wide range of application types, we performed experiments using three applications: RUBiS, RUBBoS, and Olio. These applications are widely-used cloud benchmarks and represent an eBay-like e-commerce application, a Slashdot-like bulletin board and an Amazon like book store, respectively. To emulate the users accessing the applications, we used our custom httmon workload generator\textsuperscript{25}, which supports both open- and closed-system model client behavior. For open clients, we changed the arrival rate during the course of the experiments as required to stress-test the system. For the closed case, the think-time of each client was fixed at 1 second and the number of users was varied. The change in arrival rates or number of users was made instantly.

This made it possible to meaningfully compare the system’s behavior under the two client models. As the application’s response time decreases, the throughput of closed clients approaches the value seen for open clients. Similar to the work in the previous section, the response time of a request is defined as the time that passes between sending the first byte of the request and receiving the last byte of the reply. We are mostly interested in the mean response time over 20 second intervals (4 control intervals), which are long enough to filter out measurement noise but short enough to reveal an application’s transient behavior.

Experiments were performed using variable loads to characterize the performance models’ responses to sudden workload spikes under the open- and closed-system models. The target response times used in the experiments ranged from relatively high to quite low in order to assess the performance models’ behavior in each case. The plots in this section are structured as follows. Each figure shows the results of a single experiment. The bottom x-axis represents the time elapsed since the start of the experiment. The time is divided in 5 equal intervals, each featuring a different arrival rate (for the open system model) or number of users (for the closed system model). The arrival rates or user numbers during each such interval are presented on the top x-axis. The upper graph in each figure plots the measured mean response time while the lower graph plots the capacity required over the next 5 second interval as computed by the performance model and allocated to the application. Figs. 8a to 8d show the performance of the two models when configured with different target response times under open- and closed-system models for the RUBiS application. In general, both performance models are stable with higher target response times under both system models. Moreover, both models converge to the target values quite quickly after detecting a sudden increase or decrease in workload (manifested as a rapid increase or decrease in the response time). It should be noted that both models correctly detect and adapt to the capacity requirements of both open- and closed-system models. At higher target response times, the open-system model requires more capacity than the closed system model when using comparable arrival rates and user numbers, respectively. For lower target response times, the capacity requirements for the two system models become almost identical. These situations are properly dealt with by both performance models, as shown in Fig. 8a and Fig.


\textsuperscript{25} https://github.com/cloud-control/httmon
8c for higher targets, and Fig. 8b and Fig. 8d for lower targets. We also performed experiments with Olio and RUBBoS. Due to space constraints, we only present time series plots generated using a target response time of 0.5 for these applications. As shown in Figs. 9 and 10, the queue length model does not behave well for lower target values with these applications but the inverse model remains stable. In general, the inverse model remains relatively stable irrespective of the target value under both system models. Conversely, the queue length model is less stable for lower target values under both system models.

Figure 8 Capacity allocation and response times for RUBiS under open and closed system models with 0.5s and 1.5s target response times

Figure 9 Capacity allocation and response times for RUBBoS under open and closed system models with a 0.5s target response time
In general, the inverse model remains relatively stable irrespective of the target value under both system models. Conversely, the queue length model is less stable for lower target values under both system models. The time series results show that the models behave as intended.

The performance models’ aggregate behavior over the course of each experiment was assessed using two control theoretic metrics that measure the total error observed during the system’s life span. These metrics are Integral of Squared Error (ISE) and Integral of the Absolute Error (IAE), which are computed as shown below:

\[
ISE = \sum (e(t))^2,
\]
\[
IAE = \sum |e(t)|
\]

The following table shows the errors of both the inverse and the queue-length models for RUBiS.

For Olio and RUBBoS, the table below shows similar measures but for a target response time of only 0.5 seconds.
Under the open system model, both the ISE and IAE for the queue length model are smaller than those for the inverse model when the target value is relatively high. The opposite is true for lower target values. This implies that in an open system, the queue length model is slightly preferable for higher target response times while the inverse model is preferable for lower target values. Under the closed system model, the inverse model’s ISE and IAE error values are lower than those for the queue length model irrespective of the target response time. The inverse model is thus preferable for closed systems.

Our experiments showed that both models perform well with relatively high response time targets under both closed and open-system models. However, the inverse model performs better for lower response time targets. More specifically, our results yielded the following key findings:

1) Both performance models properly predict the capacity required for both open- and closed-system models.
2) Both performance models are more stable when the response time target is relatively high. However, the inverse model is more stable than the queue-length model for lower targets.
3) The inverse model is more stable under closed system model than under open system model for lower targets.
4) Both models reach stability very quickly (i.e. within 40 seconds) after detecting a change in the system.

As in most cases, lower response time targets are usually needed, we have integrated the inverse model vertical autoscaler in CactoOpt.

2. CACTO Opt in Action

In this section we describe the features integrated and tested in the final stable release of CactoOpt released in June 2016.

Placement and Migrations decisions take into account the static resource allocation values and predefined affinity and anti-affinity constraints associated with application types (e.g., no more than one LCCSD Molpro job per physical server), network topology (e.g., no migrations between different clusters / network segments), and heterogeneity of the infrastructure (e.g., no migrations between servers with different CPU microarchitectures). On the other hand, Horizontal Scaling decisions depend on the aggregated performance of an application layer (e.g., Gamification for DataPlay application).

In general decisions regarding different types of optimisation actions are independent, i.e., one optimisation action does not directly cause another optimisation action. The only exception is the Horizontal Scaling action (scale-out) that triggers Placement decision for a new virtual machine.

Since the Placement and Migration decisions do not consider application performance and operate on the nominal capacity of virtual machines together with the Scaling decisions they capture optimizations taking into account the topology of the datacenters, the configuration of the VMs, and the application performance. The application level metrics are measured and optimized by scaling up and down the amount of resource allocated to the VMs. These decisions are passed to the optimization algorithms as either new placement decisions (in the case of horizontal scaling).
3. VISUALISATION OF THE RESULTS

In order to enable the user of CactoOpt to visualize optimization runs, we have designed and built a visualization tool for CactoOpt decisions and the datacenter. The tool is built using Python, and D3.js which is a JavaScript library for producing dynamic, interactive data visualizations in web browsers. Making use of SVG, HTML5, and CSS standards.

The interface of the visualizer is shown in the following figure. The interface shows the datacenter configuration with the physical machines and VMs mapped on them. The heterogeneity is captured by the size of boxes. The colors represent how loaded is a VM w.r.t. the amount of resources allocated, where red is extremely loaded, and green is lightly loaded.

The visualizer takes as input the configurations of the datacenter at a given point in time, and visualizes the VM movement to enable a datacenter operator to view how the decisions were made by CactoOpt through time. It presents the datacenter operator with the ability to see graphs summarizing of the state of the datacenter as shown in figures 12, 13, and 14 for example.

![Visualizer interface](image-url)
Figure 12 Graphs showing statistics on CPU bounds at a step.

Figure 13 Graphs showing statistics on the number of empty machines across the time of the simulation.
Figure 14 Graph showing the RAM utilization for a given step of the simulation.
IV. FUTURE PLANS

There are clear plans for the future of CactoOpt, these can be summarized as following.

- UMU is committed to providing support for the open sourced source-code of CactoOpt as both a standalone tool, and as part of the CACTOS toolkits for a suitable duration after the life-cycle of the project.
- UMU, ULM and DCU are part of a newly starting H2020 project, RECAP, extending on many of the results obtained in CACTOS. While CactoOpt as a component will not be part of the new project, the new management tools developed within RECAP will build on the results of CactoOpt.
- A spin-off based on Causa, the internal engine of CactoOpt optimization, is now in the last phase of planning.
- Parts of the algorithms developed within CACTOS on autoscaling are planned for integration in the commercial offerings of Elastisys AB.