CLOUDFARM: An Elastic Cloud Platform with Flexible and Adaptive Resource Management

Vladimir Nikolov
Inst. of Inform. Resource Management
University of Ulm
D-89069 Ulm, Germany

Steffen Kächele, Franz J. Hauck
Inst. of Distributed Systems
University of Ulm
D-89069 Ulm, Germany

Dieter Rautenbach
Inst. of Optimization and OR
University of Ulm
D-89069 Ulm, Germany

***Abstract*—Elasticity is a key feature of current cloud computing platforms. Dependent on their demand tenants can dynamically scale up and down their applications. To increase their revenue, cloud providers are used to over-provision their clusters, but they still have to reserve capacity to avoid that services get unresponsive and cause SLO violation during bursts. In this paper, we propose CLOUDFARM, a PaaS architecture with an adaptive SLO-based resource management mechanism. It introduces new flexible SLAs backed with a respective development model and management interface for end-user services. According to their SLAs and the price tenants pay, services can be selectively downgraded to overcome short-term peaks, e.g. while preparing for scale-out. Providers can deploy services optimistically and thus maximize their data center utilization and revenue.**

I. Introduction

With cloud computing, many unrelated applications of different tenants run on a shared third-party infrastructure. Applications can exploit elasticity, i.e. they can additionally demand or release resources. Technically this requires a cloud provider to measure current and to estimate future workloads of applications. When resources are allocated to applications too lavishly the provider wastes potential profits, since spare resources can be further leased. However, when resources are not sufficient during unexpected bursts (i.e. due to breaking news), services may become unresponsive. As there is no perfect estimator that predicts future workloads, cloud providers are currently obliged to reserve spare capacity for services, in the hope that they can handle spikes and bursts caused by unusual events. Spare capacity on both their nodes and in their cluster as a whole, however, hinders providers to fully utilize their clusters.

In this paper, we present a new SLO-based resource management for cloud platforms. Dependent on the price a tenant pays, a new set of SLOs allows that applications and services run degraded for a limited time period. For tenants these SLOs are obviously cheaper. Providers in contrary get a possibility to bridge short-time peaks or to prepare for further provisioning (e.g. scale-out, migration) causing additional start-up time when a burst persists. We propose an application model in which applications have *different modes*. Each mode provides a different QoS level and consumes different amount of resources. Our integrated resource scheduler executes applications in adequate modes to fully utilize computing nodes but also avoids overload situations. It distributes resources adaptively dependent on the SLAs customers have contracted with the provider and the actual application behavior. In our approach, we use priority-driven capacity reservation and sharing mechanisms known from soft real-time theory. Customers' applications with expensive contracts will be degraded rarely, whereas cheaper contracts are used to balance spikes. The contribution of this paper is twofold: (i) We introduce a real-time–based resource management for PaaS platforms. (ii) In combination with a new set of SLOs we propose an algorithm to overcome short-time workload peaks. Providers thus can over-provision resources even more since now they have the means to balance bottleneck situations very quickly.

The remainder of this paper is organized as follows: Before Section III introduces CLOUDFARM, Section II provides the background about the technology we use. Section IV presents related work. Finally, we conclude in Section V.

This research has been supported by DFG under Grant No. HA2207/8-1 (ARTOS) and BMBF under Grant No. 01/H13003 (MyThOS)

II. Background

Within our CLOUDFARM architecture, we use OSGi as a component-based application model that we extend by both resource management and cloud features.

Initially, OSGi supports software components (i.e. *bundles*) and manages their life-cycle using a standardized model [18]. Bundles can be installed, updated, and removed at runtime without needing to restart the entire platform. Thus, the OSGi operation model perfectly matches long running server applications that should remain reconfigurable and updatable. Bundles can contain libraries, applications or services which can be shared along with other components. A central service registry allows for service provisioning while code dependencies are transparently managed by the platform.

A. COSCA

COSCA [11] is our Java-based PaaS platform that can host multiple OSGi-based applications on shared third-party hardware. For deployment and management, it supports a concept of applications that are composed of multiple bundles. As deploying unrelated applications on a single node requires separation, the COSCA platform provides isolated OSGi-like environments for each application, so called *Virtual Frameworks* (VF). These are very lightweight and aware of the mapping of bundles to applications (i.e. they only wire bundles of the same application). Since multiple frameworks can run in the same JVM, creating a VF generates a very low footprint. COSCA can thus host multiple applications on a single host without using an additional IaaS layer that causes further

overhead. To separate applications, the platform uses different approaches on both module and service layers.

COSCA frameworks on multiple nodes form a distributed PaaS system. It is the distributed character and the fact that applications can be split across multiple COSCA instances that enables the platform to handle applications whose hardware requirements exceed the resources of one physical node, e.g., when the node does not have enough memory. This allows us to address fine-grained load balancing and scalability on component level [12]. For that purpose a COSCA application does not consist only of a set of OSGi bundles, but includes a detailed application description. This specifies an application address, firewall settings and stickiness settings for network routing. It can link system services such as a (distributed) file system and configure mount points and number of file system replicas. The deployment descriptor can also influence scaling behavior of single components. For on-demand configuration, a web interface assists in automatically creating and altering application descriptors.

During deployment of an application, COSCA analyses interdependencies between bundles and clusters them in a way that the coupling between clusters is minimized. When two components share code directly on module-level aside of service calls, COSCA treats them as one unit during distribution. For stateful components, COSCA has stickiness interaction policies. In particular, different clusters are not allowed to have shared state, but have to interact solely through services. Each cluster is then deployed widely independent from other clusters.

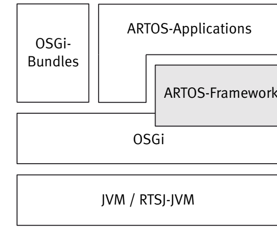


Fig. 1: ARTOS Architecture

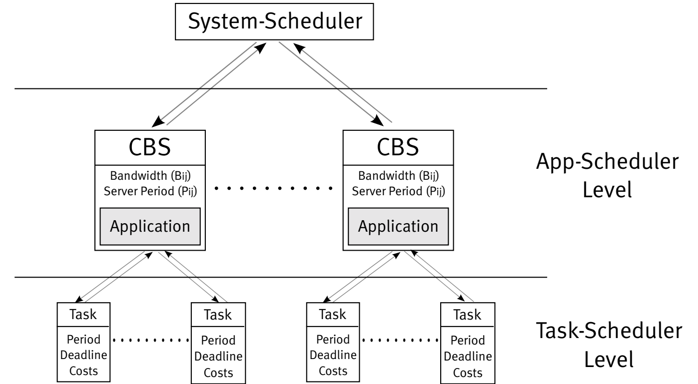


Fig. 2: ARTOS Scheduling Model

B. ARTOS

ARTOS [17] extends OSGi with a resource-centric runtime model which allows fine-grained planning and distribution of computational capacities. Its goal is to investigate scheduling and adaptation mechanisms for platforms that have to cope with a dynamic set of *unknown* applications. Our prototype supports applications with soft real-time requirements and uses RTSJ.[1] Fig. 1 depicts our framework architecture. Beside support for traditional OSGi bundles, the framework provides a special model for applications that are subject to the ARTOS resource management. Our model allows a platform-controlled adaptation on application level to accomplish optimal resource provisioning and application runtime performance. This is essential especially for overload situations where resources have to be traded in a deliberate and fine-grained manner. Furthermore, ARTOS observes long-term behavior of applications, in order to suppress recurring resource reconfigurations caused by cyclic computational bursts for example.

a) ARTOS application model: The primary aspect of ARTOS is that applications support multiple predefined *operating modes* with different computational complexity and resource consumption. In fact, we assume that at any time there is a dynamic set of n concurrently running applications $A_1,\ A_2, ..., A_n$ and for each application A_i there is a set of modes $M_{i,1}, M_{i,2}, ..., M_{i,m_i} \in \mathcal{M}_i$. Each application further consists of a set of tasks which may also implement different modes (e.g. with different periods, deadlines and costs). In that manner, an application mode manifests as a configuration of task modes which can be switched dynamically. In general, a higher application mode has a higher demand of computational resources $R_{i,j}$ (e.g. CPU costs) but therein the application delivers a better quality — e.g. a better resolution of a video codec or a higher amount of information provided by a website. The delivered quality is formalized by application-specific utility functions $\mathrm{U}_i(M_{i,j})$ given by the application developers. Each function is normalized as a relation of quality benefits between the different application modes. Each application A_i contains a mode $M_{i,0}$ with $R_{i,0} = 0$ and $\mathrm{U}_i(M_{i,0}) = 0$ which corresponds to the application being temporarily stopped.[2] Adding such a mode ensures that there is always an optimal resource distribution. For further discussion on conceptual and technical mechanisms for the realization of operating modes, we refer to [17]. Furthermore, applications can be weighted by users or the framework itself with an importance factor a_i. These factors affect most obviously each applications prominence during the resource distribution process.

b) ARTOS resource management model: ARTOS involves a three-stage hierarchical scheduling model, which is depicted in Figure 2. An application-local *Task-Scheduler* activates tasks[3] via a sophisticated priority shifting mechanism implementing an Earliest Deadline First (EDF) policy. The Task-Scheduler monitors tasks' actual costs and gathers statistical information about their execution, e.g. jitter, deviation, burst occurrence and potential burst cycles. This information is then fed to a higher-level *Application-Scheduler*.

[1] Real-Time Specification for Java

[2] Application tasks are blocked in a consistent state with infinite deadlines.

[3] Currently we assume periodic tasks, but aperiodic tasks can be handled too by the use of periodic execution servers (e.g. deferrable servers).

Each separate application is encapsulated by a *Constant Bandwidth Server* (CBS) [2] and thereby isolated from all other applications. A CBS emulates a virtual CPU with lower speed that serves a set of tasks. In ARTOS each CBS (application) receives a fraction $B_{i,j}$ (*bandwidth*) of global resources corresponding to the currently estimated demands of the hosted tasks for a certain interval of time $P_{i,j}$ (*server period*). The server period is computed as the super-period (*lcm*) of tasks periods for the currently active application mode. Accordingly, the bandwidth is computed as $B_{i,j} = R_{i,j}/P_{i,j}$, i.e. the interpolated and aggregated task costs per server period. Given a particular CBS bandwidth and the estimated task costs we are able to perform application-local feasibility analysis based on a (virtual) processor demand estimation within $P_{i,j}$. In this way, the platform can even detect unfulfillable requirements of the tasks and cancel the respective application mode if required.

The Application Scheduler activates the CBS according to their server periods following again an EDF policy. When the reserved budget of a CBS is exhausted, the server is preempted giving room for servers that still have capacities. On this way, bottlenecks are kept localized to their origins and do not affect unrelated applications. However, inflating applications are not condemned to immediately suffer from weaker performance. When reserved budgets are not consumed, spare resources are automatically reallocated to current hot spots by a capacity sharing mechanism (CASH) [6]. Moreover, CASH delays coarse-grained reconfigurations until total resource exhaustion.

In case of a permanent overload condition, sharing spare capacities is not sufficient. A *System-Scheduler* has to compute a new global resource distribution. For a given set of applications and their finite sets of modes, we establish the resource distribution by merely identifying the optimal configuration of active application modes according to their estimated bandwidth requirements $B_{i,j}$. In extreme cases, ARTOS may even deactivate applications. This can be described as an optimization problem which aims to maximize the gained utility so that the total system resource boundary R is preserved. The problem statement [17] is generally summarized as:

$$\begin{aligned}\text{Find a selection}\quad & J = \{M_{1,j_1}, M_{2,j_2}, ..., M_{n,j_n}\},\\ \text{maximizing}\quad & \textstyle\sum_{i=1}^{n} a_i \cdot \mathrm{U}_i(M_{i,j_i}),\\ \text{subject to}\quad & \textstyle\sum_{i=1}^{n} B_{i,j_i} \leq R.\end{aligned}$$

Our actual knapsack-based solution is independent of the nature of the utility functions and always calculates an optimal solution in a single run. At each iteration step the algorithm recurs over every feasible combination of modes (including the zero-modes $M_{i,0}$) and chooses a partial solution with maximal utility for the next step. The complexity of the entire solution depends on the number of applications and their modes.

Two tables, u and J, are used as data structures for the computation. For $i \in \{1, 2, ..., n\}$, $r \in \{0, 1, ..., R\}$ and $M_{i,j_i} \in \mathcal{M}_i$ let

$$\mathrm{u}(i, r) = \sum_{i=1}^{n} a_i \cdot \mathrm{U}_i(M_{i,j_i})) \tag{1}$$

denote the maximum utility such that $B_{1,j_1} + B_{2,j_2} + \ldots + B_{i,j_i} \leq r$. Let further

$$\mathrm{J}(i, r) = \{M_{1,j_1}, M_{2,j_2}, ..., M_{i,j_i}\} \tag{2}$$

be an optimal selection satisfying the given constraints. Table $\mathrm{u}(i, r)$ vividly contains the maximum possible utility for the first i applications and a total system resource boundary of r, as $\mathrm{J}(i, r)$ holds the selection which led to the gained maximum utility.

The tables are initialized with $\mathrm{u}(0, r) = 0$ and $\mathrm{J}(0, r) = \emptyset$ for all $r = 0, 1, ..., R$. For $i \in \{1, 2, ..., n\}$ and $r \in \{0, 1, ..., R\}$ the following recursion applies:

$$\mathrm{u}(i, r) = \max_{M_{i,j} \in \mathcal{M}_i} \{\mathrm{u}(i-1, r - B_{i,j}) + a_i \cdot \mathrm{U}_i(M_{i,j})\} \tag{3}$$

while with

$$\mathrm{J}(i, r) = \mathrm{J}(i-1, r - B_{i,j}) \cup \{M_{i,j}\} \tag{4}$$

an optimum realizing selection is given. Checking if $r - B_{i,j}$ is non-negative and therefore still resides within the tables ensures observation of the constraint $\sum_{i=1}^{n} B_{i,j_i} \leq R$. This definition allows for the formulation of the algorithm in pseudo code (see Algorithm 1).

Algorithm 1 OPTIMIZATION ALGORITHM

```
1:  for r = 0 → R do
2:      u[0][r] ← 0, J[0][r] ← ∅
3:  end for
4:  for i = 1 → n do
5:      u[i][0] ← 0
6:      for r = 1 → R do
7:          max ← 0
8:          for M_{i,j} ∈ 𝓜_i do
9:              b ← u[i − 1][r − B_{i,j}] + a_i · U_i(M_{i,j})
10:             if b > max then
11:                 max ← b
12:                 J[i][r] ← J[i − 1][r − B_{i,j}] ∪ {M_{i,j}}
13:             end if
14:         end for
15:         u[i][r] ← max
16:     end for
17: end for
18: return J[n][R]
```

In our experiments we used precomputed utility values for application modes, which were normalized and delivered all at once to the optimization algorithm. On this way, no further invocations of $\mathrm{U}_i(M_{i,j_i})$ were necessary during optimization.

As the knapsack problem is known to be NP-complete, no algorithm which computes the exact solution can be expected to run in polynomial time. However, dynamic programming provides a pseudo-polynomial time algorithm. Since the number of computed values in $\mathrm{u}(i, r)$ is $n * (R + 1)$ and for every entry there are at most $m_{max} = \max_{i \in \{1,2,..,n\}}\{|\mathcal{M}_i|\}$ modes, the described algorithm runs in pseudo-polynomial time. However, if all involved numerical values are kept within reasonable ranges, this should not constitute a serious problem. For example, the value denoting the total system resource boundary R can be set to any non-negative integer value. It is obviously beneficial for the computing time to keep R — and therefore the resource requirements $B_{i,j}$ for the modes of the applications — small. The increase of computing time with a larger R was evaluated in [17].

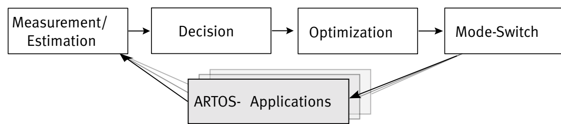
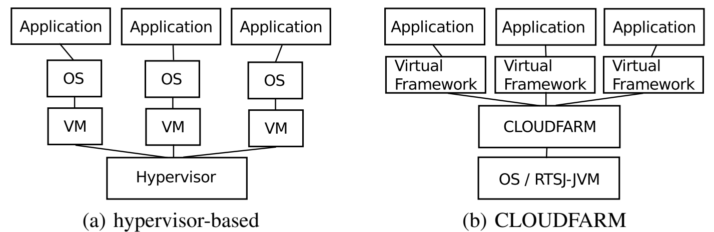


Fig. 3: System-Scheduler Feedback Loop

Figure 3 depicts the function of the System-Scheduler in a feedback loop which periodically observes the current resource distribution and workload in the platform. In certain situations, when total available resources R are overstrained or the platform configuration changes—e.g. due to modified user preferences, application arrival and departure, available spare capacities for potential higher modes—the System-Scheduler may decide to compute a new modes configuration. This is done by solving the previously described problem online via the knapsack-based algorithm. The result of the optimization is the selection of operating modes J to which applications have to be dynamically switched following an activation policy. Conversely, if the platform has spare resources $R_{slack} = R - \sum_{i=1}^{n} B_{i,j_i}$, the System-Scheduler examines the applications for potential modes $M_{i,j} \in \mathcal{M}_i$ with a higher utility and resource demands that will fit into R_{slack}. As any "better" mode will be automatically considered by the optimization algorithm, the System-Scheduler merely has to recompute a new optimal configuration J in order to exploit the slack.

c) Implications: Our proposed application management model enables an optimal resource distribution which produces the best possible overall quality in the platform with respect to available resources and actual demands. Especially in overload conditions applications can be degraded pro-actively and in a controlled manner to fit to the platforms available resources. Giving the platform enough space to calibrate its scheduling parameters according to theory deadline misses can be avoided. In reality tasks jitter and the system has to dynamically adapt to it. When for example a new application is activated its demands are first approximated. Thereby, the platform selectively activates different application modes $M_{i,j}$ until they are estimated well enough, but this may even be delayed if bottlenecks occur. A good estimation means for example that the deviation of the estimated task costs converges to a stable value which covers a desired amount actual task costs plus a dynamic jitter-dependent security buffer (see [17]). After initial approximation a new optimal configuration is realized in the platform with the gathered knowledge.

ARTOS involves several approximation mechanisms for applications resource demands, e.g. exponential smoothing, standard deviation or a quantile applied for the measured task costs, each of them defining a set of peculiar parameters. Adjustments of these factors have intense implications on the approximation quality with respect to jitter sensitivity, adaptation speed, burst detection etc. Since cost approximations directly affect resource reservations they strongly impact on the overall system performance. The online trade-off between the several approximation results needs to be examined in future work. However, the important point is that any of the used techniques tries to cover a certain percentage of tasks measured costs in order to detect changing or abnormal application behaviour. The latter denotes a hint for the potential need of a system reconfiguration.



(a) hypervisor-based (b) CLOUDFARM

Fig. 4: Resource Management

III. CLOUDFARM'S RESOURCE MANAGEMENT

CLOUDFARM combines aspects of ARTOS and COSCA allowing us to increase the utilization of cloud-computing nodes. The use of mechanisms known from soft–real-time systems helps us in both, monitoring and controlling resources of cloud applications.

a) Framework Model: Application encapsulation using VMs provides a coarse-grained approximation of different application profiles purposed to meet applications' dynamic resource demands and requirements (c.f. Fig. 4a). Unlike a hypervisor-based approach, CLOUDFARM introduces a further abstraction level by using Virtual Frameworks for each application (c.f. Fig. 4b). Such a deployment provides two major advantages. First, it is very lightweight compared to a typical virtual machine approach and thus provides us the necessary flexibility for fast reconfigurations. Second, due to a PaaS set-up, it provides the possibility to involve applications during degradation (e.g. by switching the video codec). In our architecture, from the resource scheduling point of view, VFs are entities subjected to resource reservations and have multiple modes $M_{i,j}$, just like ARTOS applications (see Section II-B). Thus, we can automatically monitor and control resource reservations on VF granularity. When the platform detects a bottleneck, we can use two strategies to overcome short-term peaks. (i) Our component-based approach enables rewiring of components and can thus select a component with lower resources consumption. (ii) Alternatively, we can also switch execution mode of a single component on degradation. In the latter, only one or more key components have to support execution modes influencing the whole application. We also propagate decisions from the bottom resource management layer to the application as events to give the opportunity to perform further platform reconfiguration if necessary. Furthermore, the resource management mechanisms in CLOUDFARM adopts merely the one of ARTOS as described in Section II-B.

b) SLOs Metrics and Mappings: SLOs are part to Service Level Agreements (SLAs) and name objectively measurable conditions for services such as system response time and availability. When monitoring resources, cloud platforms typically use metrics from the operating system to determine node workload (CPU load in percent, length of run queue, etc.). These are, however, coarse-grained average values for a given time period without the guarantee that the provider does not violate SLAs during spikes. To our best knowledge there is no reliable and standardized mapping between (SLO-defined) cloud-metrics (e.g. availability and response time) and runtime parameters of the underlying platform. First of all, such map-

pings are application and platform specific, i.e. a mere increase of global resources does not necessarily lead to a reliable increase of application performance and SLO-conformity. Even with a higher amount of available resources their distribution still establishes in an uncontrolled and unpredictable best-effort manner [4]. Second, cloud platforms lack abstract definitions of execution and performance models on application and system-level respectively. There is no way for the system to determine if an intervention, e.g. a resource reconfiguration, gains a positive effect on application performance, let alone to control that effect in a fine-grained manner for occurring hot-spots. With our proposed resource-management approach for CLOUDFARM we fill that gap through a further abstraction level and metrics. It gives cloud providers the opportunity to (prematurely) react on and control SLO-violations. Thus, a provider is able to optimize its resource utilization and costs for individual compute nodes and the overall data center as well. In the following we will give some examples of performance metrics and their mapping to runtime parameters supported by our proposed model.

In CLOUDFARM, we combine cloud SLOs with our proposed application modes where they can define both the quality level of an application and non-functional properties. *Quality level* specifies the time an application runs in a specific mode (e.g. at least 90% of the time in non-degraded mode $M_{i,j}$). Additionally, tenants can choose from different pricing models (e.g. bronze, silver, gold) at deployment, where applications running in a higher level will be degraded later, e.g. they are rated with a higher weight factor a_i. *Non-functional properties* define parameters such as response time (e.g. below 100 ms) and availability (e.g. 99 %). Our resource management algorithm uses these values to calculate degradation, elasticity and migration decisions.

c) Discussion: In cloud computing clusters, we see scenarios for both local and cluster-wide resource management. In situations where multiple applications of different tenants are running on the same host, providers usually have to leave spare capacity in order to handle unexpected peak loads and bursts. CLOUDFARM can manage local resources of computing nodes and can transiently degrade application to lower execution modes with regard to their SLA level. If a peak load persists, it uses more heavy-weight features such as migration or scale-out. The resource usage in cloud platforms *globally* also varies. In order to utilize spare capacity when there is low demand (e.g. during night), we can deploy applications with cheap SLAs running with low priority (a_i). These applications utilize spare capacity but are very likely degraded for absorbing peaks. When there are bottlenecks, we can degrade these applications even to $M_{i,0}$. Thus, they will be stopped, but keep their state for later resumption when capacity is available again.

d) Case study: response time optimization: In our case study, we use a web server hosting web pages that might be delivered in various quality levels regarding their embedded advertisements [1]. In a lower degraded mode, personalization or advertisements could be disabled leading to lower request processing costs and prices for the tenants respectively. Certainly, running in degraded modes will decrease the revenues of the tenant but in consequence the tenant can use a cheaper SLA on deployment, e.g. an SLA that guarantees an 80% execution without degradation. An according SLA would involve a clause about *response time*, e.g. the provider guarantees a maximal processing time for each website request of Q milliseconds.

Existing mechanisms (c.f. Section IV) do not guarantee a strict but a median response time observance, and they mostly focus on effects like system responsiveness and noise sensitivity due to control parameter adjustments. In our example we present a simple mechanism derived from real-time theory, which allows us to test for strict SLA-compliance e.g. on each request arrival or departure. Given a concrete application resource budged, e.g. R_{app} = *application CBS budget*, the test can be repeated until a service level is found in which all pending requests can meet the deadlines dictated by the response time SLA. If all tests fail, another strategy can be used e.g. to selectively reject requests.

From a real-time system's point of view, the scenario can be modeled as a dynamic set of n aperiodic tasks (requests), with unpredictable (sporadic) inter-arrival times r_i, measured average costs $c_{i,j}$ (for each processing mode j) and relative deadlines $d_i = r_i + Q$. For simplicity we assume that the requests are processed in FIFO order and all have the same priorities, but reordering according to any eligibility metric is also obvious. According to this model, *feasibility* of the current task set denotes compliance with the SLA's response-time clause, since a deadline miss is equal to SLA violation for the respective request. A simple feasibility test, e.g. based on Buttazzo's *instantaneous load* [5] definition, would check for exceeding 100% CPU utilization on certain load step points t, i.e. on *inter-arrival* of new requests:

$$\rho(t) = \max_i \rho_i(t), \text{ with } \rho_i(t) = \frac{\sum_{d_k \leq d_i} c_k(t)}{d_i - t}.$$

Hereby, $c_k(t)$ denotes the remaining computation time of task k, with a deadline less or equal than the one of task i. Since the test is done on each new request arrival, an overload condition (i.e. $\rho(t) \geq R_{app}$) is detected before an actual SLA violation. Now tasks can be switched to lower modes to fit optimally into R_{app}.

Depending on the inter-arrival rate of requests and their actual costs in different modes, a point may be reached where degradation cannot avoid SLA violation. We want to remind that our technique helps to smooth transient spikes until e.g. a new application instance is spawned. In case of a total overload we can even stop applications temporarily, selectively violating SLAs related to availability.

e) Case study: multimedia streaming application: Our application modes can also *directly influence algorithms* in services, e.g. video encoding quality in a streaming server. Running in a high mode, it can use a highly efficient video codec whereas on degradation it can either change parameters of the codec (e.g. the resolution or the frame rate) accordingly or switch to another video codec consuming lesser resources.

A video encoder typically processes chunks of sampled data periodically within an encoding loop. Depending on the peculiarity of the data, the complexity of the encoding process may vary. Consequently, this may lead to varying processing costs (jitter) for each encoding period. If jitter is small enough and resources are sufficient, these variations may cause slight variations in the quality of the generated picture (due to a

limited encoding buffer per data chunk). Otherwise, if jitter is high and encoding complexity increases rapidly (e.g. a stronger movement in the video scene), in the absence of complexity degradation mechanisms and fine-grained resource control this may lead to frame drops possibly violating a potential frame rate SLA-clause for the application. In fact, processing costs can increase even in a way that they become unfulfillable for the desired encoding loop duration and according frame rate.

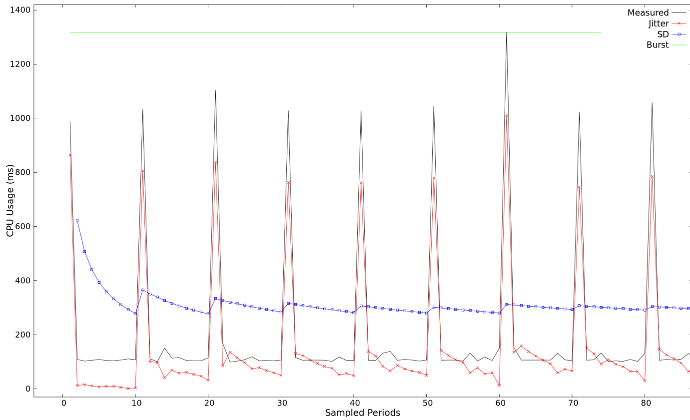


Fig. 5: Bursting Application Sample

Resource estimation within our proposed scheduling model in Section II-B continuously computes standard deviation (SD) of task costs for a number of preceding task periods (history). In our experiments we used a history depth of 1024 samples, which results in a relatively stable but sluggish approximation of tasks costs. Figure 5 shows an example of an artificial periodically bursting application. It depicts measured and computed values for tasks demands, standard deviation and jitter. Standard deviation is a measure for tasks resource demands which are interpolated and aggregated as $R_{i,j}$ for a given CBS period. For burst detection we use a 2σ margin which covers 95.45 percent of the expected task execution costs, if these costs are distributed normally. Of course tasks costs do not have to be distributed normally, but here we use the doubled standard deviation to detect outliers from a relatively constant resource consumption. If the actually measured task costs exceed the latest SD-value for one or more task periods, then we have detected a potential application burst. For example our artificial application in Figure 5 bursts every 10 periods with tenfold costs. If a burst persists, our System-Scheduler may decide to degrade the application to a lower quality level, i.e. to resize the video picture to a lower resolution though preserving the frame rate predefined by the SLA-clause. This in turn may give another VFs and running applications hosted on the same platform node the opportunity to scale up their quality (see Section II-B). On the contrary, depending on the priority a_i of the video streaming application, the scheduler may *pin* the application to the actual bursting mode, in order to give the according Task-Scheduler a chance to update its approximation. This in turn may lead to degradation of another applications running on the same node.

If an application burst recurs periodically (e.g. a periodic movement within an video-observed manufacturing line), CLOUDFARM is able to detect the burst period time span t based on a combination of *FFT*- and *autocorrelation*-based algorithms integrated within our scheduling framework. In that case, the System-Scheduler can decide if it should perform coarse-grained reconfigurations on each burst occurrence, following tasks real resource demands for an optimal resource distribution. If otherwise the burst occurrence rate t violates a tolerated intermediate reconfiguration time, the scheduler may decide to use an upper bound of the burst costs (e.g. "Burst" in Figure 5) for application bandwidth computation, in order to suppress further cyclic reconfigurations. The latter case would prefer *configuration stability* at the cost of sub-optimally utilized resources. However, since we have capacity sharing (CASH) enabled on CBS-level, conservative reservations covering burst values would be compensated and the accrued slack resources between the bursts would be balanced across all applications.

IV. Related work

Virtual machines provide one way for resource management and isolation (i.e. IaaS). Cloud providers such as Amazon and Rackspace offer a variety of instance types with different CPU, memory and I/O performances. To react to changes in load, different auto-scale mechanisms can spawn new VM instances in a pre-defined size. As prediction of the size of a VM is not an easy job, Sedaghat et al. [19] propose a mechanism to replace a current set of virtual machines with a different set to optimize the price/performance ratio. Compared to CLOUDFARM these approaches are coarse-grained. Since spawning a new virtual machine is time and resource consuming, it should be done rarely. In consequence, providers have to leave enough spare capacity to be able to react to peaks in load, even if these are transient and have short duration. performed Some hypervisors (e.g. VMWare ESXI) even provide resource management by dynamic activation of VMs to different *share* levels. When resources are insufficient, an administrator may specify that a certain VM may use a higher resource proportion as another less important one. Although this model is not as flexible as the proposed one and is based on statically predefined values for different resource levels, it constitutes a possible abstraction model for dynamic resource management. Amazon EC2 Spot Instances provide a way to run virtual machines depending on resources available in the cluster. As Amazon may terminate such an instance, it leads to a complicated app model. Customers have to use checkpointing [20] or anytime algorithms which results in a limited set of use cases. Furthermore, apps have no confidence about their reached quality, whereas in our mode-based approach the platform is able to dictate certain application quality levels.

Macias et al. [16] propose several rules for maximizing revenue for cloud providers by establishing a flow between market and resource layers. Amongst others they name variable prices (as a function of the offer/demand), resource over-provisioning, and selective SLA violation. In the latter, the provider can selectively violate some of the SLA for minimizing the economic impact, certainly, with a conflict between short-term benefit and mid-term losses. Goudarzi et al. propose an heuristic algorithm for SLA-based resource allocation for multi-tier applications in cloud computing [9]. They use processing, memory requirements, and communication resources to optimize resource allocation and total profit of the system. For management purposes, they introduce a Bronze and Gold SLA, where only the latter provides guaranteed response time.

Andrzejak et al. [3] developed a probabilistic model that answers the question of how to bed given SLA constraints. Unexpected termination of cloud applications, moreover, leads to a complicated application model. Again, customers have to checkpoint their applications leading to specialized use cases (e.g. worker nodes in Map Reduce [7]) and uncertain quality.

Zhu et al. [21] propose a control-theory–based approach for adapting *controllable parameters* of cloud applications online via a feedback loop. This method is reciprocal to our approach since we trade both utility and resource demands of application modes against each other in order to produce the best overall resource configuration. Zhu et al. further neglect overheads typically arising from continuous parameter changes (e.g. data re-encoding), while our approach tends to delay reconfigurations as long as possible via a resource sharing algorithm. *Imprecise Computations* (IC) [15] is a soft real-time scheduling model with tasks consisting of mandatory and optional parts. In general, IC-based solutions also suffer from configuration instability as in every reservation period the execution of optional parts is re-decided.

A similar work to our proposed scheduling model has been done in the scope of the ACTORS FP7 European project [8]. While their work focuses on multi-core scheduling, our resource management investigates the implications of different resource approximation techniques on configuration stability aiming at seldom application reconfigurations. Besides, we target at the automatic detection of cyclic application bursts in order to suppress cyclic reconfigurations.

Web content adaptation for response time optimization is not a novel idea. Realization mechanims have been already discussed by Abdelzaher et al. [1]. A control theoretical approach QACO [13] distributes a dynamically adjusted resource *quota* optimally among a number of pending requests. The result is a best-quality configuration of request service levels observing a response time set-point. However, control steps are solely based on an averaged input variable without consideration of job deadlines. Since processing and response time fluctuations can compensate each other the algorithm is unable to guarantee strict but a mean observance of the response time SLO. In [14] Klein et al. examine adaptive adjustment of controller parameters which decide upon the execution of optional application parts in varying load and resource contention situations. They show that while a more steady controller better observes a desired response time, the system recovers more slowly from load peaks.

V. Conclusion

Although today's cloud providers use over-provisioning, they have to reserve spare capacity to be able to react to peaks or bursts. In this paper, we propose SLO-based resource management that helps avoiding SLA violation even in fully utilized clusters. Our mode-based execution model can adapt the execution of applications with regard to both, the free capacity in the cluster and the SLO they have. CLOUDFARM manages resource reservations for different applications in an elastic cloud platform according to their actual resource demands and predefined SLOs.

In future work, we plan to further support tasks with real-time requirements and to address availability in cloud SLOs by adding support to handle faults [10].

References

[1] T. F. Abdelzaher and N. T. Bhatti. Web Content Adaptation to Improve Server Overload Behavior. *Computer Networks*, 31, 1999.

[2] L. Abeni and G. C. Buttazzo. Integrating Multimedia Applications in Hard Real-Time Systems. In *IEEE Real-Time Systems Symposium*, 1998.

[3] A. Andrzejak, D. Kondo, and S. Yi. Decision model for cloud computing under sla constraints. In *IEEE Int. Symp. on Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2010.

[4] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, Randy H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the Clouds: A Berkeley View of Cloud Computing. 2009.

[5] G. C. Buttazzo and J. A. Stankovic. Adding robustness in dynamic preemptive scheduling. In *Resp. Comp. Sys.: Steps Toward Fault-Tolerant Real-Time Sys.*, volume 297. Springer US, 1995.

[6] M. Caccamo, G. Buttazzo, and L. Sha. Capacity sharing for overrun control. In *Proc. of the 21st IEEE Real-Time Sys. Symp.—RTSS*, 2000.

[7] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Comm. ACM*, 51(1), 2008.

[8] J. Eker, E. Bini, G. Buttazzo, S. Schorr, R Guerra, G. Fohler, K.-E. Arzen, V. Romero, and C Scordino. Resource Management on Multicore Systems: the ACTORS approach. *IEEE Micro*, 31, 2011.

[9] H. Goudarzi and M. Pedram. Multi-dimensional SLA-based resource allocation for multi-tier cloud computing systems. In *Proc. of the 2011 IEEE 4th Int. Conf. on Cloud Comp.* IEEE, 2011.

[10] F. J. Hauck, S. Kächele, J. Domaschka, and C. Spann. The COSCA PaaS platform: on the way to flexible and dependable cloud computing. In *Proc. of the 1st Europ. Workshop on Dep. Cloud Comp.*, EWDCC '12. ACM, 2012.

[11] S. Kächele, J. Domaschka, and F. J. Hauck. COSCA: an easy-to-use component-based PaaS cloud system for common applications. In *Proc. of the First Int. Workshop on Cloud Comp. Platforms*, CloudCP '11. ACM, 2011.

[12] S. Kächele and F. J. Hauck. Component-based scalability for cloud applications. In *Proc. of the 3rd Int. Workshop on Cloud Data and Platforms*, CloudDP'13. ACM, 2013.

[13] J. Kim, S. Elnikety, Y. He, S. Hwang, and S. Ren. Qaco: Exploiting partial execution in web servers. In *Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference*. ACM, 2013.

[14] C. Klein, M. Maggio, K.-E. én, and F. Hernández-Rodriguez. Brownout: Building more robust cloud applications. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014.

[15] J. W. S. Liu, W. K. Shih, K.-J. Lin, R. Bettati, and J.-Y. Chung. Imprecise computations. *Proc. of The IEEE*, 82, 1994.

[16] M. Macias, J.O. Fito, and J. Guitart. Rule-based SLA management for revenue maximisation in cloud computing markets. In *Int. Conf. on Network and Service Management (CNSM)*, 2010.

[17] V. Nikolov, M. Matousek, D. Rautenbach, L. Draque Penso, and F. J. Hauck. ARTOS: System model and optimization algorithm. techn. rep., 2012.

[18] OSGi Alliance. OSGi service platform core spec. 4.3, 2011.

[19] M. Sedaghat, F. Hernandez-Rodriguez, and E. Elmroth. A virtual machine re-packing approach to the horizontal vs. vertical elasticity trade-off for cloud autoscaling. In *Proc. of the 2013 ACM Cloud and Autonomic Comp. Conf.*, CAC '13. ACM, 2013.

[20] Sangho Yi, D. Kondo, and A Andrzejak. Reducing costs of spot instances via checkpointing in the amazon elastic compute cloud. In *2010 IEEE 3rd International Conference on Cloud Computing*, 2010.

[21] Q. Zhu and G. Agrawal. Resource provisioning with budget constraints for adaptive applications in cloud environments. *IEEE Transactions on Services Computing*, 5(4), 2012.