
http://www.diva-portal.org

Postprint

This is the accepted version of a paper presented at *International Conference on Cloud and Autonomic Computing (ICCAC), SEP 21-25, 2015, Boston, MA, USA.*.

Citation for the original published paper:

Tomas, L., Tordsson, J., Vazquez, C., Moreno, G. (2015)
Reducing Noisy-Neighbor Impact with a Fuzzy Affinity-Aware Scheduler.
In: *2015 INTERNATIONAL CONFERENCE ON CLOUD AND AUTONOMIC COMPUTING (ICCAC)* (pp. 33-44). New York: IEEE Computer Society
http://dx.doi.org/10.1109/ICCAC.2015.14

N.B. When citing this work, cite the original published paper.

Permanent link to this version:
http://urn.kb.se/resolve?urn=urn:nbn:se:umu:diva-109498

Reducing Noisy-Neighbor Impact with a Fuzzy Affinity-Aware Scheduler

Luis Tomás and Johan Tordsson
Department of Computing Science
Umeå University, Sweden
Email: {luis,tordsson}@cs.umu.se

Carlos Vázquez and Ginés Moreno
Department of Computing Systems
University of Castilla-La Mancha, Spain
Email: {Carlos.Vazquez,Gines.Moreno}@uclm.es

***Abstract*—Overbooking techniques have been proven efficient to increase overall utilization of cloud datacenters. However, overbooking may also degrade applications performance as (at least) some applications need to share physical resources such as CPU or memory. Consequently, interference may increase among the virtual machines that share resources, the so called noisy neighbors effect. We present an affinity-aware scheduler to reduce the impact of such interference. A fuzzy logic engine accounts for the uncertainty in these environments and estimates which CPU cores are currently more suitable for each incoming application. This helps the scheduler make virtual machine to physical resource mapping decisions, also known as vcpu pinning. An experimental evaluation based on a combination of interactive services and batch applications confirms that our affinity-aware fuzzy scheduler reduces the interference among applications, enabling more predictable performance and consequently safer overbooking.**

***Keywords*—Cloud Computing, Clustering, Fuzzy Logic Programming, In-Server Scheduling, Noisy Neighbor, Overbooking**

I. Introduction

Cloud datacenter resources are not being used in an efficient way, mainly due to one of its main features: elasticity and fast provisioning. These cloud characteristics complicate capacity management due to uncertainty about future user resource needs. Therefore, if these needs are overestimated, the resource utilization of the datacenter becomes low. In contrast, underestimation of resource requirements can lead to performance degradation and even crashes. Cloud providers are commonly conservative and accept poor resource utilization ratios for the sake of safe execution and performance assurance. In fact, reported utilization ratios are well below half the total available capacity [1], leading not only to wasted energy but also to revenue loss at the providers side.

Resource overbooking has been proven as a suitable solution to increase resource utilization ratios [2], specially when users overprovision their resource requests and/or datacenters suffer from the Virtual Machine (VM) sprawl phenomenon. Resource overbooking is a well known technique, previously applied e.g., in network bandwidth multiplexing. In a cloud context, overbooking is mainly based on allocating more VMs than the actual number of physical resources in a datacenter, taking advantage of the fact that not all users use all the capacity they requested and/or they do not use it all the time [3].

Even in absence of overbooking, VMs in a datacenter share some resources, such as CPU, memory, or cache hierarchies. This may lead to VMs affecting or being affected by other co-located VMs. These interferences between VMs are commonly known as the *noisy neighbor* problem [4]. This effect is aggravated in overbooked systems as resources are more exhaustively used, and due to the higher consolidation, more resources need to be shared among VMs. This in turns leads to degradation and to unstable applications performance, and consequently to less predictable overall system behavior.

To alleviate the noisy neighbor problem, the choice of which VMs should share resources must be taken with care, including an analysis of the impact VMs have on each other. This is a multi-dimensional (CPU, memory, cache, etc.) optimization problem commonly solved using heuristics [5] [6]. Due to the complexity of the problem, as well as the uncertainty about future status of CPU cores and application needs, we take a fuzzy logic approach. Fuzzy logic is a highly expressive, natural, and efficient way to deal with uncertainty and approximate reasoning, and we previously applied this approach to cloud admission control with good results [7].

To address the challenge of noisy neighbors, we introduce a fuzzy-logic affinity-aware engine that helps the VM scheduler to decide *in-server scheduling*, i.e., deciding which VMs can share CPU cores without affecting each other significantly. This fuzzy-logic engine is mainly based on the classification of both the status of cores (i.e., CPU intensive, memory intensive, network intensive, or a mix of them) and the expected VM needs. Then it decides which cores are more suitable to host a particular VM, i.e., have high affinity. The enabling mechanism for this in-server scheduling is the KVM core pinning that limits (but not completely removes) the impact that one VM may have on others by restricting the specific physical cores that each VM is allowed to use.

The experimental results show the benefits of using our proposed fuzzy affinity-aware server scheduler in a mix of interactive and non-interactive applications, of which some need to meet certain deadlines while others need to maintain specific throughputs or response times. The VM interference is reduced and consequently a more constant utilization ratio is achieved regardless of the workload mix, as well as more stable, predictable applications performance. This in turn leads to safer overbooking decisions, enabling cloud datacenter operators to increase their revenue by making a more efficient use of their resources.

The structure of this paper is as follows. In Section II the background scenario and the related works are presented. In

Section III the main contributions of this work are detailed. Next, Section IV presents our experimental results. Finally, Section V concludes the paper and outlines directions for further research.

II. Background and Related Work

As reviewed in [3], in addition to elasticity, there are other factors contributing to poor resource utilization at clouds datacenters, such as predefined VM sizes [8] or the user tendency to overestimate their needs [9]. Resource overbooking has been studied as a solution to mitigate this resource utilization problem [2]. However, deciding what is the suitable amount of overbooking, given the current set of applications, is a really difficult task, and is mostly approached by predictions. Furthermore, these predictions may be wrong or not accurate enough, leading to either performance degradation (and Service Level Objective (SLO) violations) or missed opportunities to increase usage.

There are some works focusing on this overbooking ratio. For instance, Urgaonkar et al. [10] try to safety overbook cluster resources, guaranteeing applications performance using feedback control and assuming that users are capable of providing information regarding the tolerable overbooking level of their applications. However, this information is strongly coupled to the underlying physical infrastructure, as well as to other co-located applications. In our case, to account for this uncertainty and possible prediction errors, we presented a fuzzy admission control [11] that decides whether a new VM can be accepted without relying on user information about how tolerable their applications are to overbooking. The steps performed are: (1) predict the future status of the data center; (2) evaluate the possible impact of accepting a new VM by using a fuzzy logic engine that accounts for the uncertainty about future events [7]; and finally (3) take the admission decision based on the acceptable level of risk (threshold) of the datacenter at the current time. The risk thresholds are obtained through a distributed set of PID controllers that adjusts their own risk threshold based on current vs. target utilization levels (for more details see [11]).

There are similar approaches that also change the overbooking ratios over time in a transparent manner to the users, such as the works presented in [12] and [13]. However, once the applications are admitted, they do not provide any mechanism to deal with resource shortages due to mispredictions or incorrect overbooking decisions, which may lead to performance degradation, specially for latency-critical applications. To tackle this problem we allowed our framework to adjust the target utilization levels (and consequently the overbooking pressure) based on applications performance input, while at the same time, some applications can use the brownout approach [14] to mitigate short-term problems until the system adapts to the new situation [15].

Once the applications are accepted, the next step is to decide how to co-locate them in a way that the interferences among them are minimized. This so called in-server scheduling has a great impact on the performance of the deployed applications [16], specially in overbooked environments where (at least) some of the physical resources (e.g., physical cores) must be shared among different applications. It is of great importance to limit the impact that noisy neighbors on other running VMs, specially when the overbooking actions were not enough accurate.

Our previous framework either relayed either on the KVM scheduler to perform the physical CPU (pcpu) sharing among VMs [11], or made use of the KVM pinning functionality to provide some isolation among VMs, with quality of service (QoS) differentiation purposes [17]. In the latter, we present a method to isolate some VMs from others (and consequently limiting their impact on each other) inside an overbooked server by efficiently using KVM virtual to physical CPU pinning. As a result, we are able to provide different overbooking and isolation degrees inside a single server, thus enabling QoS differentiation for different applications. However, some VMs could still interfere with each other, specially the ones with low QoS requirements.

There are also other works that try to ensure certain QoS through SLAs. For instance, Beloglazov et al. [18], Bobroff et al. [19], and the Sandpiper engine [20] present different methods to detect overload situations and trigger migrations to resolve them. However, once again they are focused on detecting and resolving the problem rather that avoiding it. In addition, they do not target the noisy neighbor problem at the in-server scheduler level. Another work focusing on VM interference and QoS is presented by Nathuji et al. [21], who propose to provision underutilized resources to the applications that require it to meet their QoS needs. However this approach requires that some capacity is left unused in the servers to be able to add it later to the applications that suffers from interferences. By contrast, in our work we are targeting an overbooked environment where the nominal capacity allocated already is larger than the real available capacity. Thus, there is no extra (unassigned) capacity to allocate to VMs if they have interference problems.

The main focus of this work is therefore focused on extending the previously presented overbooking framework so that VMs has a more stable and predictable performance by limiting the impact that other co-located VMs may have on them. There are several attempts in the literature to tackle the VM-interference problem. A multivariate probabilistic model based on VM (anti)affinity rules with the aim of avoiding wrong co-location actions was presented by He et al. [22]. Similarly, Meng et al. [23] proposed a joint VM provisioning approach based on estimating aggregate VM capacity requirements, but assuming accurate predictions about future workload. Mars et al. [16] present the bubble-up approach, where they propose to predict the performance interference of co-located applications by both measuring the pressure that an application generates on the memory subsystems and how much that application is affected by different levels of pressure. A similar approach is presented by Delimitrou et al. in [5], who propose a scheduler that classifies and allocates the incoming applications based on the profiled and expected interference with other already running VMs. Although they pursued the same target – reducing applications interferences – and follow a similar approach – differentiating applications and isolating them from their antagonist – they focus on the scheduling between servers, not inside the servers. Therefore these two approaches complement our work.

There are works based on detecting performance degra-

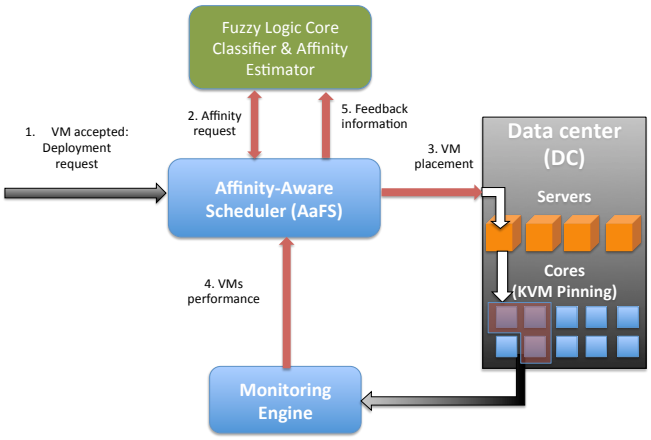


Figure 1: VM deployment overview.

dation due to VM interference and that use migrations to alleviate the problem when possible. One such workare used by Zhang et al. [6], where CPI (Cicle Per Instruction) performance metrics were used to detect anomalous behaviors and then recursively throttle the VMs that are believed to be the possible source of the interference problem, i.e., the VM affecting the other(s). However, this solution does not avoid the problem but tries to recover once it occurs, and it can impact applications performance due to the throttling.

Finally, an in-depth analysis of co-location problems leading to QoS violations is presented by Leverich et al. in [24], drawing conclusions about why some collocations may have bigger impacts than others. Based on these conclusions, they propose to use the Borrowed Virtual Time (BVT) scheduler instead of the standard Completely Fair Scheduler (CFS) used by Linux systems. Unlike the work presented in [24], we build our solution on top of an standard hypervisor, in our case KVM. This motivates our decision of providing performance isolation through hypervisor tools, in this case KVM core pinning. Although simple, KVM pinning can help to avoid poor performance by limiting the impact that some applications may have on others. For example, it could avoid that some VMs use specific physical cores where a latency-critical VM is running.

III. Affinity-Aware In-Server Scheduler

The co-location problem is multi-dimensional (note VMs are not only using CPU but also memory, and network) and is usually addressed by using heuristics. In a similar approach as used in our work on admission control [11], we decide to tackle the server scheduling and co-location problems with fuzzy logic.

As depicted in Figure 1, the *Affinity-aware Fuzzy logic assisted Scheduler (AaFS)* chooses both the server where the VM is going to be allocated, and assigns the physical cores inside the chosen server to the new accepted VMs. These choices are based on the affinity scores provided by the *Fuzzy Logic Core Classifier & Affinity Estimator* module. This fuzzy engine classifies the cores based on their recent behavior and estimates the affinity scores between the cores and the VM to be allocated. With this information, the scheduler can reduce interferences by co-locating the VMs based on their affinity values, so that their performance is impacted as little as possible.

These two components are the main novelty of this work. As the system is designed to work on overbooked environments, the main objective is to generate good co-location plans that reduce the VM(s) noisy neighbor problem by mapping the VM virtual cores (vcpus) to specific physical cores (pcpus), so that they are only shared by some selected VMs. To generate a good mapping, the status of cores is continuously classified (i.e., clustered) depending on their recent behavior with respect to CPU, memory, and network usage. Based on this information, the fuzzy engine identifies the (group of) cores with higher affinity scores for the incoming VM. Finally the AaFS assigns the VM to the proposed pcpus and provides feedback to the fuzzy engine to readjust the affinity scores over time based on the obtained performance.

A more in-depth explanation about how the AaFS uses the fuzzy engine and its affinity information is presented next, as well as how the feedback information is injected into the fuzzy engine. After that, we detail how the fuzzy logic classification and affinities are calculated.

A. AaFS Scheduler: Feedback Affinity Loop

The AaFS module is in charge of the scheduling decisions, i.e., decides *where* and *how* the VMs are placed onto physical servers. This scheduling process is divided in two steps. First, the server *where* the VM will be allocated is chosen based on overbooking ratios of each server – each server can tolerate a different level of overbooking depending on what VMs it provisions. The full details of how to calculate the overbooking factor (based on application performance deviations) is presented in our previous work [11]. Once the server is decided, a second step determines *how* the vcpus are pinned to pcpus, i.e., determines which specific physical resources the VM can use inside the selected server.

As depicted in Figure 1, the AaFS Scheduler is assisted by the above described fuzzy engine to perform this second step. From this engine, the AsFS gets information about the affinity between the incoming VM and the pcpus in their current status. The cores with higher affinity (i.e., affinity values over a specific threshold) are selected and the VM vcpus are pinned to the given cores (pcpus) by using the KVM pinning functionality [25]. Thanks to the pinning, the impact that a VM may have on others is reduced, as it cannot steal CPU time for other VMs that are pinned to different pcpus. Notably, interference problems cannot be fully avoided with KVM pinning as the memory and some cache levels are still shared. Furthermore, if the pinning is not performed in an efficient way, the interference could be even higher as some antagonist [6] VMs may be co-located and they are not allowed to use other pcpus even if there exists idle cores. Moreover, due to the uncertainty about future resource needs, as well as due to the dynamic nature of cloud environments, the same settings cannot work properly all the time. Therefore, there is a need of autonomous adaptation to changing situations. To achieve this, we have implemented a feedback loop between the AaFS, in this case the applications performance (Key Performance

Indicators, KPIs), and the fuzzy engine. By doing this, we enable the possibility of adjusting the affinity values based on application performance.

The fuzzy engine offers two knobs to adjust the affinity outputs, named `increase` and `decrease` (detailed in next subsection) that have opposite behaviors. By calling `increase`, the output affinity values are increased and in turns that leads to a higher consolidation of the VMs onto fewer pcpus. Reversely, if `decrease` is invoked, the affinity values are decreased, leading to a more balance distribution of the VMs over the available pcpus. To make an efficient use of this steering mechanism we need to dynamically decide when the VMs consolidation is too high, and hence some applications have problems to keep their performance. Or by contrast, if the load is well balanced and the applications are achieving their KPIs without problems, we could consolidate all VMs a bit more to allow additional VMs to be provisioned.

Based on these two tuning knobs in the fuzzy engine, we have designed a controller that decides if the affinity values need to be increased, decreased, or if the current configuration is suitable. The final decision about the action to be taken at each control interval is based on the status of the running applications, as well as on how balanced the cores usage is, as outlined in Algorithm 1. In order to obtain the applications performance, a controller inside the applications computes a **matching value** m_i, that expresses how the application is performing for one of its KPIs:

$$m_i = 1 - r_i/\bar{r_i}, \tag{1}$$

$$m_i = 1 - \bar{t_i}/t_i, \tag{2}$$

where r_i is the maximum response time over the last control interval and $\bar{r_i}$ is the target response time, and t_i and $\bar{t_i}$ are the minimum and target throughput for the throughput oriented applications, respectively. These matching values are positive if the application maintains the desired KPI and become negative if application is suffering performance degradation. Note that these matching values abstract application performance indicators, such as target response time or throughput, from the infrastructure.

On the other hand, to measure whether the CPU core usage is balanced, we have designed a simple heuristic mostly based on their average usage and standard deviation. Mainly, we denote the core usage to be unbalanced if one of the following is true:

- The standard deviation of cores usage is bigger that an specific threshold, in our case 20 percentage-points.
- There are cores fully utilized (utilization is above a certain, parameterizable threshold, in our case 80%), as well as almost idle cores (below a certain, parameterizable threshold, in our case 20%).
- More than 20% of the cores have usage ratios that deviate from the average usage ratio by more than 10 percentage-points. Both these thresholds are also parameterizable.

The *feedback control loop* implemented between the server and the fuzzy engine adjusts the affinity values over time as

Algorithm 1 Affinity Values Controller

Configuration parameters: $duration$, update interval
w_bad, weight of applications with performance problems
$w\_veryBad$, weight of applications with large performance problems
t_good, threshold for minimum number of applications with good performance
t_bad, threshold for allowed number of applications with performance problems

1: **while** true **do**
2: $app\_good \leftarrow$ number of applications with good performance
3: $app\_bad \leftarrow$ number of applications with bad performance
4: $app\_veryBad \leftarrow$ number of applications with very bad performance
5: $unbalanced \leftarrow$ true if the cores usage is unbalanced
6: **if** $app\_good > (w\_bad * app\_bad + w\_veryBad * app\_veryBad)$ and $app\_good > t\_good$ **then**
7: INCREASE()
8: **else**
9: **if** $unbalanced$ and ($app\_veryBad$ or $app\_bad > t\_bad$) **then**
10: DECREASE()
11: **end if**
12: **end if**
13: sleep for $duration$
14: **end while**

detailed in Algorithm 1. In the first part of the algorithm information about application performance and usage balance is obtained. Then, to decide if the affinity values can be increased, it is checked whether the number of applications keeping their performance (i.e., has positive matching values) is much larger than the number of applications having problems to maintain it. In that case, and if the amount of applications with good performance is large enough, the affinity values are increased in order to achieve a higher consolidation. Otherwise, no increase actions are required.

On the other hand, as detailed in the last part of Algorithm 1, to decide if the affinity values must be decreased, it is first checked if the cores usage is unbalanced. Otherwise no actions are taken as the application problems do not stem from the consolidation but are due to the system having too many VMs in overall. By contrast, if the usage is unbalanced, then it is checked if there is a VM with a really bad performance (i.e., high negative matching values), or several VMs with bad performance (i.e., negative matching values). In such a case affinity values are decreased, that in turn would yield better performance. Otherwise, if only a few VMs have slight performance degradation, affinity values are not decreased.

B. Fuzzy Logic Core Classifier & Affinity Estimator

Here, we explain the *Fuzzy Logic Core Classifier & Affinity Estimator* engine in more detail. It is implemented by using the *Fuzzy LOgic Programming Environment for Research* $\mathcal{FLOPER}$ [1] [26], and a preliminary version is described in [27]. The fuzzy programs are coded with a variant of the popular logic language Prolog [28] augmented with expressive resources inspired by fuzzy logic [29]. For instance, instead of managing the pair $\{true, false\}$ of classical logic, here truth

[1] http://dectau.uclm.es/floper/

$$\&_{\mathtt{P}}(x,y) \triangleq x * y \qquad |_{\mathtt{P}}(x,y) \triangleq x + y - x * y \qquad \leftarrow_{\mathtt{P}}(x,y) \triangleq \min(1, x/y)$$

$$\&_{\mathtt{G}}(x,y) \triangleq \min(x,y) \qquad |_{\mathtt{G}}(x,y) \triangleq \max\{x,y\} \qquad \leftarrow_{\mathtt{G}}(x,y) \triangleq \begin{cases} 1 & \text{if } y \leq x \\ x & \text{otherwise} \end{cases}$$

$$\&_{\mathtt{L}}(x,y) \triangleq \max(0, x+y-1) \qquad |_{\mathtt{L}}(x,y) \triangleq \min\{x+y,1\} \qquad \leftarrow_{\mathtt{L}}(x,y) \triangleq \min\{x-y+1,1\}$$

Figure 2: Fuzzy conjunction, disjunction and implication connectives from *Łukasiewicz* (pessimistic), *Gödel* (optimistic) and *Product* (realistic) logics, respectively, defined in the real unit interval.

degrees are real numbers in the unit interval $[0, 1]$, which better captures the uncertainty of the cloud environment. The engine also provides several standard connective definitions as presented in Figure 2, that have different capabilities for modeling *pessimistic*, *optimistic*, and *realistic scenarios* (respectively labeled as `L`, `G` and `P`). Other frequently used connectives are "`aver`", "`not`", "`approx`", and "`very`", defined as $aver(x,y) = (x+y)/2$, $not(x) = 1 - x$, $approx(x) = \sqrt{x}$, and $very(x) = x^2$, respectively, or the more complex operator "`over`". The latter is shown in Figure 3 and explained later in this text.

The implemented fuzzy engine offers an interface with the following five operations to the AaFS:

- *Initialize* is used to configure the fuzzy engine with the static information of the server hardware details. It has three input parameters: the number of cores of the server; the number of previous resource usage values (CPU, memory, and network) to be considered in the cores classification calculation (merged by a moving average); and the hardware server details regarding distances between each pair of cores. The latter is needed to force VMs vcpus to be close to each other when possible.
- *Actualize* updates the core usage information in the fuzzy engine. With this new information, the fuzzy system actualizes its internal database and consequently updates the information about the intensity in the usage of each resource (CPU, memory, and network) as well as the usage burstiness for each core.
- *Assign* returns the group of cores best suitable to allocate a certain VM, i.e., the group with highest affinity values. Its input parameters include *Vm*, the virtual machine to be allocated; *NCoresToAssign*, the number of cores that the VM needs; *DiscardedNodes*, an optional parameter to discard a group of cores in case they are exclusively reserved for other VM(s); and *Threshold*, a parameter to enhance the efficiency of the engine by pruning the solution search tree of solutions based on a minimum affinity threshold required.
- *Increase* and *Decrease* are both intended to modulate the truth value of the solutions given by previous function *assign*. The *increase* operation increases affinity values, i.e., makes the search for suitable cores more optimistic. Conversely, *decrease* lowers affinity values and makes the search for cores more pessimistic.

In addition to that external interfaces used by the AaFS, there is another internal function to evaluate the distance between the physical cores. This is needed at the assign call to better evaluate and find a group of cores where a multi-core VMs would perform better, not only due to VM interference but also due to possible communication patterns between VM vcpus. Thus, the notion of *distance of cores* is just a measure of the cost, in terms of time, of communicating between two cores, and that tries to account for the NUMA (Non Uniform Memory Access) architecture features of todays' processors. We provide this information to the system through the function `distance(Core1, Core2, Distance)` based on the configuration information sent at the system initialization. As an example, in our testbed, servers have 32 cores, divided in 4 groups of 8 cores. Here we use a distance between cores of 10 if they are in the same group and 16 otherwise, i.e., we specify that cost of communication among cores is 60% higher for cores that are not in the same group.

In order to return the group of cores with higher affinity with the requested VM (*Assign* function), the cores need to be first classified with respect to their behavior over time (by using the *actualize* function). We consider both the *intensity* and *burstiness* – i.e., the usage irregularity – in the resource utilization. To perform the cores status classification, we make use of several fuzzy predicates: *X_Intensive(core)* and *X_Bursty(core)*, where X is the capacity measured, i.e., CPU, memory, or network. Their returned value (truth degree) is the extent to which a certain *core* is intensive in its use of resource type X (e.g., CPU) or has a bursty behavior in its use, respectively.

Once the cores status is classified, *assign* searches and returns the group of cores that are most suitable for the incoming VM, as well as the associated affinity value (truth degree). The process used to determine whether a group of cores is adequate for a certain VM, i.e., the affinity value between a VM and a group of cores, is shown in Algorithm 2. This algorithm calculates the overall affinity between the accepted VM and a group of cores by taking into account the cores $X\_Intensive$ for each resource X (CPU, memory, and network). The affinity between a VM and a group, for a specific capacity dimension ($Cap_X\_Intensive\_Affinity$) is calculated by using fuzzy connectives, in this case the operator not, $aver$, and $over$, by Equation 3:

$$Cap_X\_Intensive\_Affinity = Not(Over(X\_Intensive(VM), X\_Intensive(Gr))) \tag{3}$$

and for $Cap_X\_Burstiness\_Affinity$ we use Equation 4, which takes into account the worst-case scenario where usage spikes in both VM and the group of cores coincide (hence the combination of $X\_Intensive$ and $X\_Burstiness$):

$$\begin{aligned} Cap_X\_Burstiness\_Affinity = \\ Not(Over(\\ Aver(X\_Intensive(VM), X\_Burstiness(VM)), \\ Aver(X\_Intensive(Gr), X\_Burstiness(Gr)))). \end{aligned} \tag{4}$$

Algorithm 2 Affinity VM vs. Group of Cores Computation

Configuration parameters: VM, input VM.
Gr, group to evaluate affinity.
$Luka$, Łukasiewicz fuzzy connective.
$Godel$, Gödel fuzzy connective.
$Product$, Product fuzzy connective.
$very$, very fuzzy connective.

1: **for each** X in $Capacity\_dimension_{\{CPU,Mem,Net\}}$ **do**
2: $Cap_X\_Intensive\_Affinity \leftarrow VM$ vs. Gr Intensive Affinity for capacity dimension X
3: $Cap_X\_Burstiness\_Affinity \leftarrow VM$ vs. Gr Burstiness Affinity for capacity dimension X
4: **end for**
5: $Aggregated\_Intensive\_Affinity \leftarrow Luka(Cap_{CPU}\_Intensive\_Affinity, \text{very}(Cap_{Mem}\_Intensive\_Affinity), Cap_{Net}\_Intensive\_Affinity)$
6: $Aggregated\_Burstiness\_Affinity \leftarrow Godel(Cap_{CPU}\_Burstiness\_Affinity, \text{very}(Cap_{Mem}\_Burstiness\_Affinity), Cap_{Net}\_Burstiness\_Affinity)$
7: $Final\_Affinity \leftarrow Product(Aggregated\_Intensive\_Affinity, Aggregated\_Burstiness\_Affinity)$
8: **return** $Final\_Affinity$

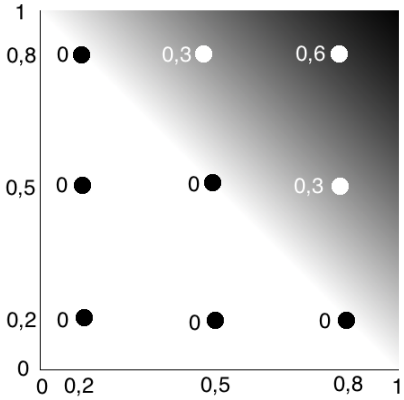


Figure 3: Behavior of connective *over* according to the expression $x@_{over}y = min\{max\{0, x + y - 1\}, 1\}$.

The aggregator *over*, refers to the possibility of overload. The expression $x@_{over}y$, being x and y the extent to which a VM and a core are using a certain resource, is 0 if the core has room for the VM. Otherwise, the value of the expression grows to 1, when the core is at its limits and the VM requests all of that resource. Figure 3 illustrates this aggregator with three VMs and three cores.

Note that after obtaining the affinities for each capacity dimension X, we combine intensive affinities with the pessimistic *Łukasiewicz* conjunction to ensure the stability of the system. On the other hand, the $X\_Burstiness$ values are combined with the optimistic *Gödel* conjunction since peaks are usually not a severe threat to the system unless they coincide and have long duration. As memory is more critical than other hardware capacities, we require a higher level of affinity for this critical resource, so we modulate it with the *very* aggregator. Finally, the values from equations 3 and 4 are merged with the realistic *product* conjunction (given by the P operation in the algorithm).

In addition to obtain the most suitable group for a specific VM, the proposed fuzzy engine enables the output affinity values to be either increased or decreased based on feedback provided by the AaFS. These output affinity values are modulated through the above described *increase* and *decrease* commands. In essence, the fuzzy engine modulates the truth degree of the *suitability* of each group through a weighted average. Empirically, we set a limit for the weight impact into the affinity outputs, only allowing a change of 20% from the base configuration. The calls to *increase* and *decrease* simply increases or decreases these weights so that the system can evolve to compute higher or lower suitability values.

These feedback functions are introduced into the system in order to gain accuracy and evolve over time based on the observed behavior. For instance, in the case that all solutions provide a very high truth degree, the datacenter can easily end up with an unbalanced core utilization. In such a case the AaFS calls *decrease*, leading to a more balance usage in the future and a more accurate pruning in the search for good enough groups for allocating VM.

IV. Evaluation

In this section we evaluate the proposed affinity-aware fuzzy assisted scheduler. First, the used testbed and workload are detailed. Next, a set of experiments are described that measure and compare the performance of our approach regarding achieved utilization and applications behavior over time.

A. Testbed and Workload

The tests are conducted on two machines, one hosting applications and one generating the workload, both in terms of number of applications (in this case VMs) and incoming workload (queries) to some of those VMs (the interactive applications). The servers are connected with a Gigabit Ethernet link. The first machine is a server consisting of a total of 32 cores (AMD Opteron™ 6272 at 2.1 GHz) and 56 GB of memory, where the applications are deployed inside VMs. The second machine is a 4-core (Intel Core™ i5 processor at 3.4 GHz) desktop with 16 GB of memory.

The workload is generated by mixing different VMs types, recreating what could be a representative cloud workload [1]. In these experiments, we follow the sand and boulders scheme [12], where there is a mix of large, long living VMs (the boulders) and short living, usually smaller VMs (the sand ones). In our case, for the boulders, we generate two large VMs (8 core, 14GB RAM each) that each run an interactive application each for the full duration of the experiment. For this we used two popular cloud benchmarks: *RUBiS* and *RUBBoS*. RUBiS [30] is an auction website benchmark modeled after eBay, while RUBBoS [31] is a bulleting board benchmark modeled after Slashdot.

To generate the incoming workload of these two applications, the number of queries that they received over time is recreating the same behavior extracted from the Wikipedia [32] traces, but scaled to the size of our system. Moreover, we chose two different (random) days and time-shifted the original workload 12 hours for RUBiS to create different trends and

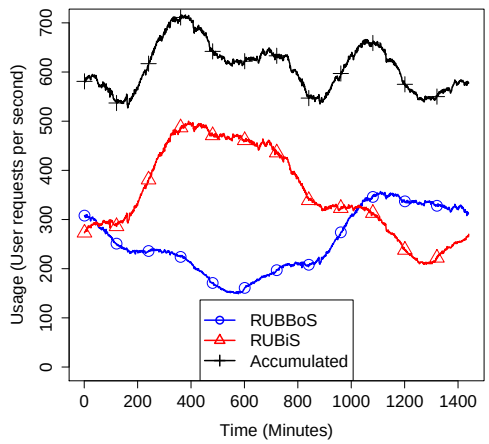


Figure 4: Workloads for interactive applications (long living VMs).

peaks (see Figure 4). By doing this we generate different daily usage patterns for each application. All client queries were generated using the `httpmon` tool[2].

For the smaller sand VMs, we generate a stream of short lived and non-periodic applications, with an arrival rate pattern that follows a Poisson distribution with $\lambda = 30$ seconds.

We generate different kind of sand VMs to have a more realistic workload, where the uncertainty about the future needs is less predictable. To this end, we build VMs running computational tasks, with highly heterogeneous and time-varying resource requirements [3]. First, to increase the uncertainty and burstiness of the system, we generate several VMs that run different shell scripts, generating burstiness in the different capacity dimensions (CPU, memory, network). Secondly, in order to have a measurable performance of the sand applications, we also create a set of VMs that solves random sudokus[3] and reports the throughput achieved over time. The sudoku VMs have more predictable performance than the ones running scripts. The different behaviors for this type of VMs are:

- *SudokuBE*: The VMs in this class need to solve a certain amount of sudokus before a given deadline. We set the deadline in such a way that the CPU requirements over time need to be around 80% of the CPU during the whole execution. As this type of VM behaves in a best effort manner, it tries to solve as many sudokus as possible. Therefore, if it runs in isolation, it uses as much CPU as available until it solves all the sudokus, with a margin of roughly 20% to the deadline.
- *Sudoku10*: unlike the best effort VMs, this VM class aims to keep a certain throughput (number of solved sudokus) over time. The set target is to solve 10 sudokus per second. Therefore, if during one time period this is not achieved, sudokus queue up, and need to be solved in the next period – in addition to the ones corresponding to the current period. This type of VM maintains an average CPU usage percentage of roughly 35% when run in isolation.
- *Sudoku20*: Same as the previous one but with higher computational requirements. The throughput to be kept is double compared to the previous one. This type of VM maintains an average CPU usage of roughly 75% when run in isolation.

These three sudoku VM classes are mixed equally in the resulting workload. However, we change the fraction of sudoku VMs to the total amount of sand applications. We perform tests where this percentage is varied between 1/3, 2/3 and 3/3. Consequently, as the number of computational intensive tasks increases, there are fewer bursty VMs, resulting in a more predictable behavior.

B. Experiment Results

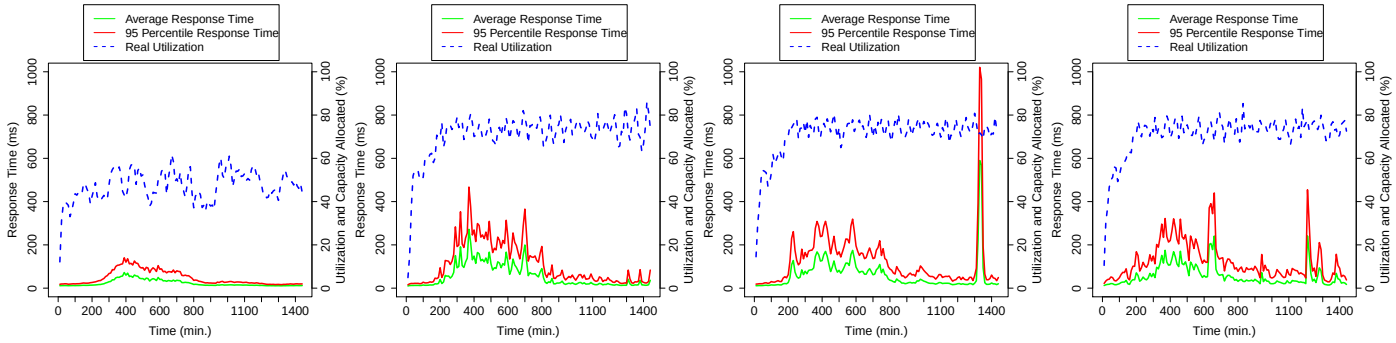
In order to evaluate the performance of the presented affinity-aware scheduler, we run different experiments with the workload above explained and compare the next 3 different techniques:

- The *standard KVM scheduler*, where by default the VMs (vcpus) are not pinned to specific cores (pcpus) and KVM is in charge of dynamically deciding which vcpu accesses which pcpu over time. This is evaluated both with and without overbooking, labeled as `Over KVM` and `No-Over KVM` in the next figures, respectively.
- A *worst-fit style pinning* scheduling algorithm that performs vcpu to pcpu pinning base on current pcpus usage, choosing the cores that are the least overbooked at the moment. This method is labeled as `Over-WF` in the following figures.
- Our proposed *affinity-aware fuzzy assisted scheduler*, labeled as `AaFS` in the figures.

We study three performance metrics metrics. First, we compare the **real utilization** achieved for each technique. This measures the actual capacity being used by all the hosted applications, which is directly related to the revenue that the cloud provider can achieve, as well as to how efficiently the hardware resources are used. Although all hardware capacity dimensions were considered in the performance evaluation, the utilization in the subsequent figures only refer to CPU usage. The other considered dimensions, memory and I/O, were also prevented from overload with admission control techniques, as explained in [11]. Due to the workload characteristics, memory and I/O were less stressed than the CPU.

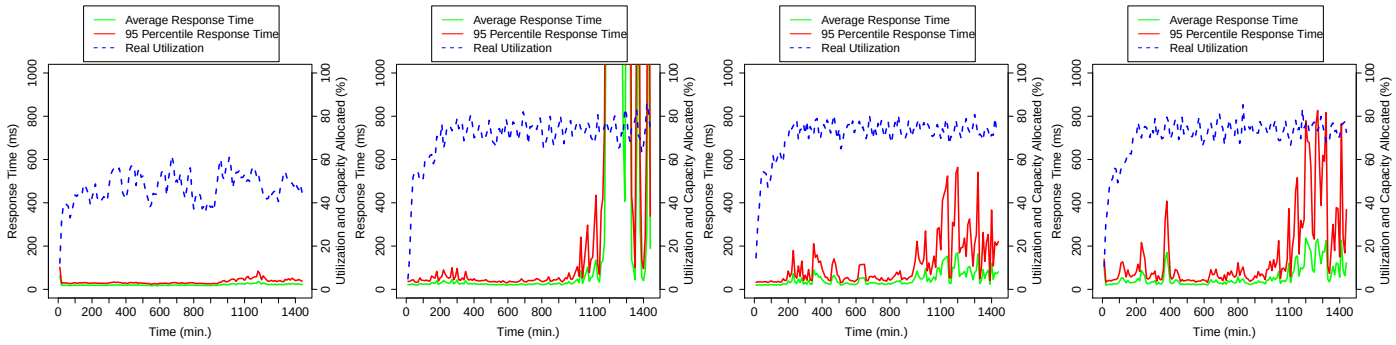
The other two metrics are related to the VMs performance. First, we measure the **average** and **95-percentile response times** of the boulder VMs. The main objective is to evaluate how these interactive applications are affected by other co-located VMs, and how the proposed scheduler helps to reduce the interferences within acceptable limits. We pay special attention to the interactive applications (in our case the boulders) as they are usually more affected by other co-located applications. They need to maintain a certain performance over time, unlike computation intensive applications that usually are not affected by a few seconds of performance interference,

[2]https://github.com/cloud-control/httpmon
[3]http://norvig.com/sudoku.html



(a) No-Over KVM (b) Over KVM (c) Over-WF (d) AaFS

Figure 5: Performance of RUBiS VM over time.



(a) No-Over KVM (b) Over KVM (c) Over-WF (d) AaFS

Figure 6: Performance of RUBBoS VM over time.

as long as they complete all computations on time. Finally, as we also want to evaluate how the sand applications are affected due to the noisy neighbor effect, we measure a number of metrics related to the performance of the sudoku VMs (i.e., sudokuBE, sudoku10, and sudoku20). These are based on the **concurrent number of sudoku VMs**, as well as on the **average** and **aggregated throughput** of all of them.

First, we compare the performance of the above-mentioned scheduling techniques with a sand application mixture composed of 2/3 of sudoku VMs and 1/3 scripts. This represent the median case where stable CPU intensive applications dominate but there is still quite some burstiness in the workload. Figure 5 shows both response times (both average and 95 percentile) for the RUBiS service when following the described Wikipedia workload and the server utilization for the whole workload – both boulders VMs plus the sands VMs. Figure 5a and 5b show the result when using the default KVM scheduler, both without and with overbooking, respectively. Figure 5c shows the result for the worst fit pinning algorithm, and finally, Figure 5d shows the results obtained for the presented affinity-aware fuzzy scheduler approach. As we can see, the achieved response times for No-Over KVM are very low, but at the expense of a rather low overall utilization as well. Utilization also presents large fluctuations based on the VMs types concurrently running. For the other three scenarios, the utilization ratio is much higher and stable – around 75%. This is also depicted in Figure 7, where a box-plot highlights the similar results obtained for the three overbooked scenarios,

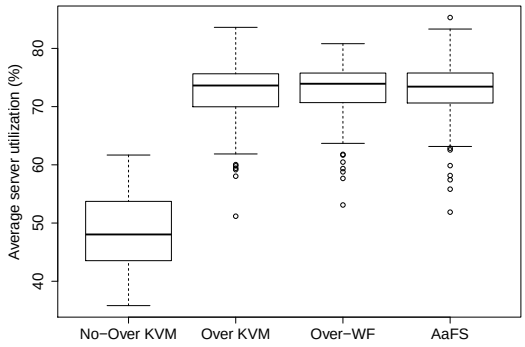
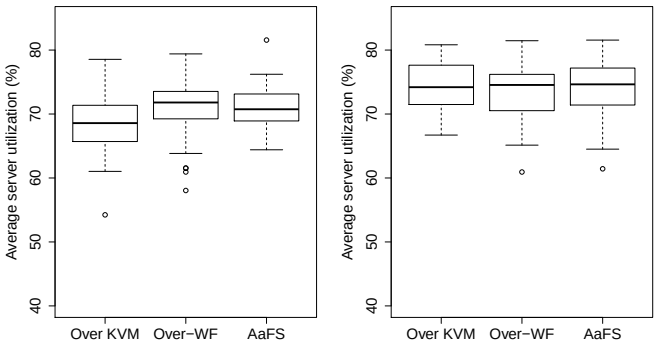


Figure 7: Utilization (2/3 of sand applications are sudoku VMs.

unlike the one without overbooking, that presents a remarkably lower utilization, as well as with higher fluctuations. Regarding response times, they are a bit higher for all overbooked scenarios too but still well below the target – 1 second. There is one sudden increase in the response time for Over-WF around minute 1300, as a result of a wrong co-location decision due to the burstiness of the co-located VM(s).

In Figure 6 the same information as in Figure 5 is shown, but for the RUBBoS application. This application presents a less linear behavior with regards to the number of requests, making it more exposed to co-location interferences. As a result, the KVM scheduler (Over KVM) is not able to maintain



(a) 100% sudoku VMs. (b) 1/3 sudoku VMs.

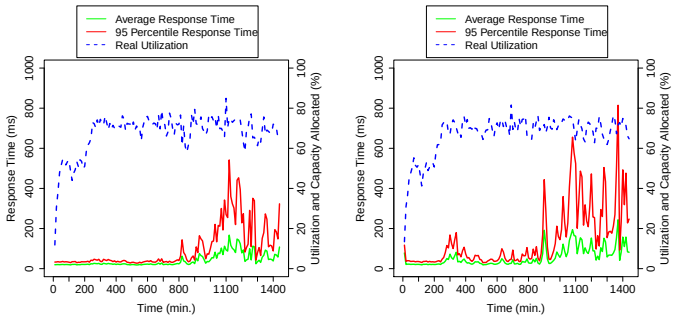
Figure 8: Usage comparison with different ratio of sudoku VMs in the sand applications.

the performance for the RUBBoS VM all the time, and there is an interval from minute 1100 onwards where the response time exceeds the 1 second target. By contrast, both pinning techniques (Over-WF and AaFS) are able to keep the performance all the time due to limiting the interferences impact through pinning decisions. Note that both keep response times below 1 second, but show larger fluctuations than for the RUBiS application.

We have also compared the performance when the amount of burstiness in the sand applications is either decreased (100% sudoku VMs) or increased (1/3 of sudoku VMs). As Figure 8 shows, the utilization is higher when there is more burstiness in the system, and it decreases otherwise as there is less opportunities for overbooking with lower burstiness. The Over-KVM is more affected by this and its utilization ratio is lower than 70% when all sand applications are sudoku VMs. On the other hand, Over-WF and AaFS present more stable performance, with less fluctuations upon changing ratio of sudoku VMs.

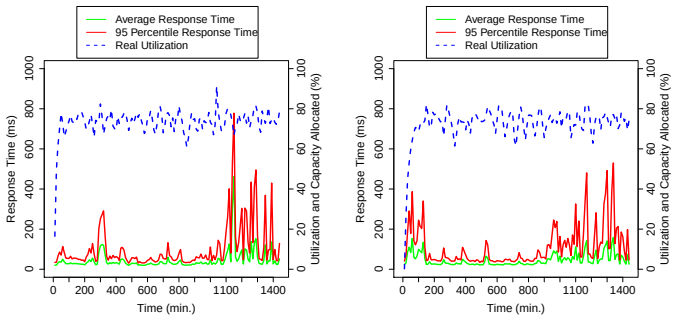
From these set of experiments, we can conclude that the KVM scheduler is not able to avoid VM interferences and is more affected by the sand mix. Therefore, in the rest of the experiments we focus on the other two approaches, Over-WF and AaFS. First, in Figures 9 and 10 we present the result obtained for the RUBBoS VM (the interactive application more affected by interference) when the sand mix made of 100% and 1/3 sudoku VMs, respectively. As before, both techniques are able to avoid large interference and keep the RUBBoS performance during all the experiment, regardless to the sand VMs mix.

Once we concluded that both Over-WF and AaFS are able to keep interactive applications (boulders) performance over time, even under different mix of sand VMs, we study the performance obtained for the sand applications for these experiments. To do that, we focus on the performance obtained by the sudoku VMs under the different sand applications mix. First, Figure 11 shows the aggregated performance obtained for the sudoku VMs for both Over-WF and AaFS under the three different sudoku mix at the sand applications (1, 2/3, and 1/3, respectively). More specifically, it shows the aggregated



(a) RUBBoS Over-WF (b) RUBBoS AaFS

Figure 9: Performance of RUBBoS VMs when all sand applications are sudoku VMs.



(a) RUBBoS Over-WF (b) RUBBoS AaFS

Figure 10: Performance of RUBBoS VM when 1/3 of sand applications are sudoku VMs.

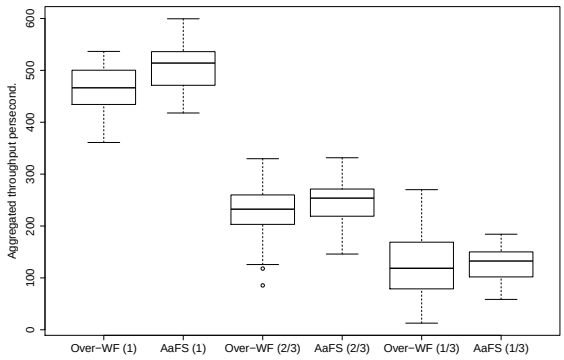


Figure 11: Aggregated performance for sudoku VMs.

number of sudoku solved per second for all the concurrently running sudoku VMs. This metric gives us a notion of the overall performance achieved by the sand applications, in particular for the first case as that includes only sudoku VMs in the sand mix. As the figure depicts, AaFS outperform Over-WF in all the cases (average values increase up to 9%), as well as presents a more stable performance (smaller boxes). This is most prominent for the 1/3 scenario, when the uncertainty is higher in the system. In that case, the standard deviation (sd) is almost 45% lower (i.e., better) for AaFS compared to Over-WF.

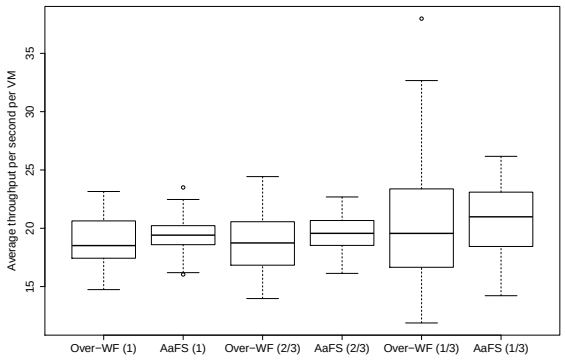


Figure 12: Average performance per sudoku VMs.

On the other hand, when we compare the average performance achieved per sudoku VM instead of comparing the aggregated performance, we observe similar results. In Figure 12, AaFS achieves higher and a more stable performance per sudoku VM for all the three cases, with average values being improved around 5%, and standard deviation of the sudoku VMs performance being reduced between 32 and 45%. Once again, the most prominent difference is for 1/3 case. It can be observed that, as the burstiness of the sand VMs mix increases, the performance obtained by the sudoku VMs fluctuates more due to the higher uncertainties of the VMs requirements and less accurate predictions. We observe that AaFS behaves remarkably better than Over-WF when the uncertainty of the system is higher (sd reduced by 45%).

In order to demonstrate other improvements that AaFs presents over Over-WF, we next show a summary of the performance achieved for the different sudoku VM types (sudoku10, sudoku20, and sudokuBE). Figure 13, Figure 14, and Figure 15 show the results obtained for the 3/3, 2/3, and 1/3 sudoku VMs cases, respectively. These figures show a comparison of the aggregated performance (plots (a) and (b)), the average performance (plots (c) and (d)), and the number of concurrently running sudokus (plots (e) and (f)).

As regards to the aggregated performance obtained per VM, a more stable performance is achieved by AaFS in all the cases (less variation in the aggregated performance over time), as well as a more proportional performance between sudoku10 and sudoku20. Ideally, the total throughput for sudoku20 should be double that for sudoku10, in absence of VM interferences and with enough resources. We also observe a slightly lower aggregated performance for AaFS with sudoku10 in the 3/3 case. However, this difference comes from the fact that Over-WF accepts more sudoku10 VMs at expenses of sudoku20 VMs, as depicted in Figure 13e and 13f. Due to this fact, we appreciate a remarkable difference in the aggregated throughput achieved by sudoku20 VMs. Similarly, when the burstiness of the sand VMs mix increases (1/3 case), Over-WF penalizes the number of accepted sudokuBE VMs in favor of accepting smaller VMs due to not consolidating the VMs when possible as AaFS does. Therefore the aggregated throughput for this type of VMs is remarkably lower than for AaFS.

When focusing on the average performance obtained per VM type, figures 13, 14, and 15 show that both Over-WF and AaFS keep the performance for sudoku10 and sudoku20 VMs in the three sand VM mix cases. The main difference is that AaFS present noticeable fewer fluctuations for sudoku20 VMs than Over-WF, which means a more stable, constant performance over time. The sd is reduced between 21.2 and 44.3% compared to Over-WF. If we focus on sudokuBE, we observe that AaFS also presents a slightly higher performance for the first two scenarios (3/3 and 2/3). Moreover, as the uncertainty of the system increases, (see Figure 15), sudokuBE are less affected by other VMs' interference and present a more stable performance over time, with a large reduction of sd: 38%.

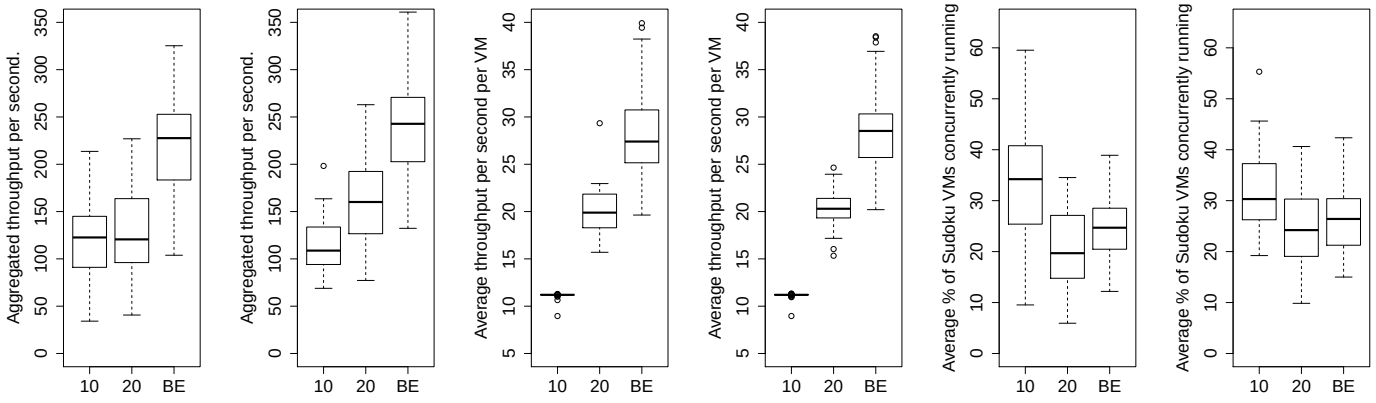
Finally, the plots with the number of concurrent sudokus over time clearly present a more balanced behavior for AaFS, where the number of sudoku VMs of each type is more balanced than for the Over-WF. It is also clear that as the percentage of sudoku VMs over the sand applications increases, the number of accepted VMs of each type gets less balanced. The main reason for this stem from the fact that as the burstiness in the system decreases, the chances for finding good co-locations and VMs consolidation decrease too. Thus, as the smaller VMs (sudoku10) are easier to allocate, the system prefers them over to other two sudoku VM types. We see that AaFS clearly outperforms Over-WF in this aspect, and is less impacted by this problem. Unlike Over-WF, AaFS presents a well balanced mix of sudoku VM types when the percentage of sudoku VMs in the sand applications is reduced. In fact, the average number of concurrent VMs for the three types of sudoku VMs ranges from 6% (for 2/3 case) to 42% (for 3/3 case) less for AaFS compared to Over-WF.

To sum up, AaFS provides not only a slightly higher performance, but also a more predictable and stable performance over time. This in turns helps the cloud provider to perform safer overbooking decisions, therefore providing a better performance to the users with higher utilization ratio, and limiting the impact of VM interference.

V. Conclusions and Future Work

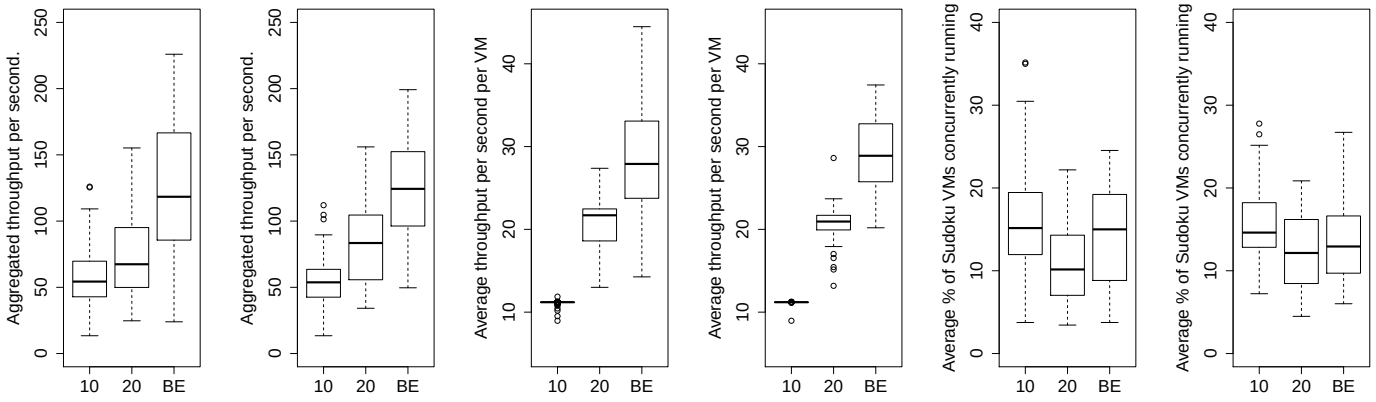
Cloud computing enables easy and fast management and provisioning of capacity, but at the same time complicates the capacity management due to uncertainty about users needs in the (near) future. In addition, the use of VM consolidation to achieve higher utilization makes VM performance more affected by other co-located VMs. Reducing this interference in case of inaccurate predictions or incorrect co-location actions is therefore required to provide more stable and predictable performance. To tackle that problem, it is essential to co-locate the VMs inside the servers so that the interference among them is limited. We present a fuzzy affinity-aware engine that helps the scheduler to decide which VMs can share which resources (in this case physical CPUs), as well as to limit the impact that VMs may have on other co-located VMs.

The proposed fuzzy affinity-aware engine is based on evaluating and classifying the server's cores depending on their usage and finding the group of core(s) that presents the best affinity with the VM to be placed. By using this information in the scheduler, together with the KVM core pinning capability, we demonstrate that interference among VMs can be reduced significantly and application performance



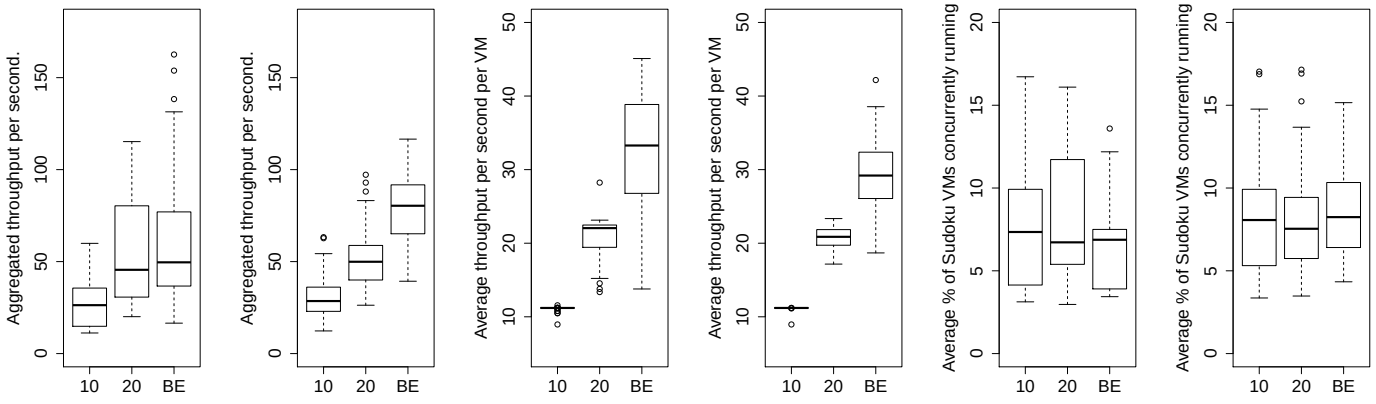
(a) Over-WF: Aggregated Performance (b) AaFS: Aggregated Performance (c) Over-WF: Average Performance (d) AaFS: Average Performance (e) Over-WF: Concurrent VMs (f) AaFS: Concurrent VMs

Figure 13: Summary of sudoku VMs performance with sand VMs consisting of only sudoku VMs (3/3).



(a) Over-WF: Aggregated Performance (b) AaFS: Aggregated Performance (c) Over-WF: Average Performance (d) AaFS: Average Performance (e) Over-WF: Concurrent VMs (f) AaFS: Concurrent VMs

Figure 14: Summary of sudoku VMs performance with sand VMs consisting of 2/3 sudoku VMs.



(a) Over-WF: Aggregated Performance (b) AaFS: Aggregated Performance (c) Over-WF: Average Performance (d) AaFS: Average Performance (e) Over-WF: Concurrent VMs (f) AaFS: Concurrent VMs

Figure 15: Summary of sudoku VMs performance with sand VMs consisting of 1/3 sudoku VMs.

thus become more stable over time compared to the existing KVM scheduling algorithm. In addition, thanks to the feedback loop between the infrastructure and the fuzzy-affinity engine, the affinity values are readjusted over time based on the achieved performance. Consequently, the proposed solution allows cloud operators to better use their infrastructure while still provisioning application according to their performance guarantees.

As future work, we plan to extend the current work by designing a more advanced reinforcement technique that could allow the fuzzy engine to better readjust the affinity values based on the performance of the previous co-location decisions. We are also interested in better categorizing and modeling the source and type of interference happening under different VMs co-location mappings, with the objective of early detecting and reacting to such undesired situations.

ACKNOWLEDGEMENTS

Financial support has been provided in part by the Swedish Research Council (VR) under contract number C0590801 for the project Cloud Control, and by the EU (FEDER) and the Spanish MINECO Ministry (*Ministerio de Economía y Competitividad*) under grant TIN2013-45732-C4-2-P.

REFERENCES

[1] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Towards understanding heterogeneous clouds at scale:Google trace analysis," Carnegie Mellon University, Pittsburgh, PA, USA, Tech. Rep. ISTC–CC–TR–12–101, Apr. 2012, http://www.istc-cc.cmu.edu/publications/papers/2012/ISTC-CC-TR-12-101.pdf.

[2] R. Householder, S. Arnold, and R. Green, "On cloud-based oversubscription," *International Journal of Engineering Trends and Technology (IJETT)*, vol. 8, no. 8, pp. 425–431, 2014, http://arxiv.org/abs/1402.4758v2.

[3] L. Tomás and J. Tordsson, "Improving Cloud Infrastructure Utilization through Overbooking," in *ACM Cloud and Autonomic Computing Conference (CAC)*, 2013.

[4] S. Verboven, K. Vanmechelen, and J. Broeckhove, "Black box scheduling for resource intensive virtual machine workloads with interference models," *Future Gener. Comput. Syst.*, vol. 29, no. 8, pp. 1871–1884, 2013.

[5] C. Delimitrou and C. Kozyrakis, "Quasar: Resource-Efficient and QoS-Aware Cluster Management," in *Intl. Conference on Architectural Support for Programming Languages and Operating Systems*, 2014.

[6] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes, "CPI2: CPU performance isolation for shared compute clusters," in *SIGOPS European Conference on Computer Systems (EuroSys)*, 2013, pp. 379–391.

[7] L. Tomás and J. Tordsson, "Cloudy with a chance of load spikes: Admission control with fuzzy risk assessments," in *6th IEEE/ACM Intl. Conference on Utility and Cloud Computing*, 2013, pp. 155–162.

[8] D. Gmach, J. Rolia, and L. Cherkasova, "Selling t-shirts and time shares in the cloud," in *Proc. of Intl. Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2012, pp. 539–546.

[9] W. Cirne and F. Berman, "A comprehensive model of the supercomputer workload," in *Proc. of Intl. Workshop on Workload Characterization*, 2001, pp. 140–148.

[10] B. Urgaonkar, P. Shenoy, and T. Roscoe, "Resource overbooking and application profiling in shared hosting platforms," in *OSDI*, 2002, pp. 239–254.

[11] L. Tomás and J. Tordsson, "An autonomic approach to risk-aware data center overbooking," *IEEE Transactions on Cloud Computing*, vol. 2, no. 3, pp. 292–305, 2014.

[12] D. Breitgand, Z. Dubitzky, A. Epstein, O. Feder, A. Glikson, I. Shapira, and G. Toffetti, "An adaptive utilization accelerator for virtualized environments," in *IEEE Intl. Conference on Cloud Engineering (IC2E)*, 2014, pp. 165–174.

[13] R. Ghosh and V. K. Naik, "Biting off safely more than you can chew: Predictive analytics for resource over-commit in iaas cloud," in *Proc. of 5th Intl. Conference on Cloud Computing*, 2012, pp. 25–32.

[14] C. Klein, M. Maggio, K.-E. Årzén, and F. Hernández-Rodríguez, "Brownout: Building more robust cloud applications," in *36th Intl. Conference on Software Engineering (ICSE)*, 2014, pp. 700–711.

[15] L. Tomás, C. Klein, J. Tordsson, and F. Hernández-Rodríguez, "The straw that broke the camel's back: safe cloud overbooking with application brownout," in *2nd IEEE Conference on Cloud and Autonomic Computing*, 2014, pp. 151–160.

[16] J. Mars, L. Tang, K. Skadron, M. Soffa, and R. Hundt, "Increasing utilization in modern warehouse-scale computers using bubble-up," *Micro, IEEE*, vol. 32, no. 3, pp. 88–99, 2012.

[17] L. Tomás and J. Tordsson, "Cloud service differentiation in overbooked data centers," in *7th IEEE/ACM Intl. Conference on Utility and Cloud Computing*, 2014, pp. 541–546.

[18] A. Beloglazov and R. Buyya, "Managing overloaded hosts for dynamic consolidation of virtual machines in cloud data centers under quality of service constraints," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 7, pp. 1366–1379, 2013.

[19] N. Bobroff, A. Kochut, and K. Beaty, "Dynamic placement of virtual machines for managing sla violations," in *10th IFIP/IEEE Intl. Symposium on Integrated Network Management (IM)*, 2007, pp. 119–128.

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

[21] R. Nathuji, A. Kansal, and A. Ghaffarkhah, "Q-clouds: Managing performance interference effects for qos-aware clouds," in *Eurosys*, 2010.

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

[23] X. Meng, C. Isci, J. Kephart, L. Zhang, E. Bouillet, and D. Pendarakis, "Efficient resource provisioning in compute clouds via VM multiplexing," in *Proc. of the Intl. Conference on Autonomic Computing (ICAC)*, 2010, pp. 11–20.

[24] J. Leverich and C. Kozyrakis, "Reconciling high server utilization and sub-millisecond quality-of-service," in *9th European Conference on Computer Systems (EuroSys)*, 2014.

[25] IBM Knowledge Centre, "Kernel Virtual Machine (KVM): Best practices for KVM," Web page at http://www-01.ibm.com/support/knowledgecenter/linuxonibm/liaat/liaatbestpractices_pdf.pdf?lang=en.

[26] G. Moreno and C. Vázquez, "Fuzzy logic programming in action with floper," *Journal of Software Engineering and Applications*, vol. 7, pp. 273–298, 2014.

[27] C. Vázquez, G. Moreno, L. Tomás, and J. Tordsson, "A cloud scheduler assisted by a fuzzy affinity-aware engine," in *IEEE International Conference on Fuzzy Systems, FUZZ-IEEE 2015, Istanbul, Turkey, August 2-5*, 2015, p. 8 (in press).

[28] J. Lloyd, *Foundations of Logic Programming*. Springer-Verlag, Berlin, 1987, 2nd edition.

[29] L. A. Zadeh, "Fuzzy logic and approximate reasoning," *Synthese*, vol. 30, pp. 407–428, 1965.

[30] RUBiS: Rice University Bidding System, Web page at http://rubis.ow2.org/, Visited 2014-01-27.

[31] RUBBoS: Bulletin Board Benchmark, Web page at http://jmob.ow2.org/rubbos.html, Visited 2014-01-27.

[32] Page view statistics for Wikimedia projects, Web page at http://dumps.wikimedia.org/other/pagecounts-raw/, Visited 2014-01-27.