Drowsy-DC: Data center power management system

Mathieu Bacou[†*], Grégoire Todeschi[†],
Alain Tchana[†], Daniel Hagimont[†]
[†]IRIT, Université de Toulouse, CNRS, Toulouse, France
{first.last}@enseeiht.fr
[†]Atos Integration, Toulouse, France

Baptiste Lepers[‡], Willy Zwaenepoel[‡]
[‡]Operating Systems Laboratory
École Polytechnique Fédérale de Lausanne
Lausanne, Vaud, Switzerland
{first.last}@epfl.ch

***Abstract*—In a modern data center (DC), a large majority of costs arise from energy consumption. The most popular technique used to mitigate this issue is virtualization and more precisely virtual machine (VM) consolidation. Although consolidation may increase server usage by about 5–10%, it is difficult to actually witness server loads greater than 50%. By analyzing the traces from our cloud provider partner, confirmed by previous research work, we have identified that some VMs have sporadic moments of data computation followed by large periods of idleness. These VMs often hinder the consolidation system which cannot further increase the energy efficiency of the DC. In this paper we propose a novel DC power management system called Drowsy-DC, which is able to identify the aforementioned VMs which have matching patterns of idleness. These VMs can thus be colocated on the same server so that their idle periods are exploited to put the server to a low power mode (suspend to RAM) until some data computation is required. While introducing a negligible overhead, our system is able to significantly improve any VM consolidation system; evaluations showed improvements up to 81% and more when compared to OpenStack Neat.**

***Index Terms*—virtualization, consolidation, long-lived mostly idle, low-power state, power consumption**

I. Introduction

Cloud platforms have proven to be able to provide very attractive prices. The enabling element of cloud computing is the *virtualization* technology. It allows a server to execute, simultaneously and in an isolated manner, multiple operating systems (OSs) called virtual machines (VMs). Reducing datacenters (DC) energy consumption is a major concern for cloud providers and an important center of interest for the research community. According to the scale at which the issue is addressed, previous works can be organized in two categories: (1) those interested in minimizing the power consumption of a single server; and (2) those which act at the DC scale, leveraging the virtualized nature of the workload.

In the first category, the holy grail is to achieve *energy proportional* servers [1]. Such a server consumes energy in exact proportion to its utilization. Thereby, a server with no load would theoretically consume no energy. In this respect, an essential goal is to independently control the power state of each hardware component (e.g. by leveraging the ACPI[1]). Subsequently, this modular system is able to adapt the energy consumed by each component according to its current solicitation. Although these solutions improve the energy efficiency for CPUs, HDDs and NICs, state-of-the-art proposals are still a long way from the ideal energy proportional machines. Therefore, some DC-level power management techniques such as VM consolidation have been proposed.

VM consolidation seeks to dynamically assign VMs to a minimum number of servers, so that empty servers — which are consequently inactive — can be transitioned to low power states (suspend to RAM or suspend to disk, respectively ACPI S3 and S4). By increasing the load of active servers, this technique also improves the energy efficiency of the DC. However, the general utilization boost is around 5–10% which makes it difficult to actually witness server loads greater than 50% for even the best adapted workloads [1, 2, 3, 4]. This is explained by several reasons. First, VM consolidation is a bin-packing problem which is NP-complete. Second, the resource partitioning depends on the resource type. For instance, unlike CPU which is time-shared, memory is space-shared between the VMs and is not preempted. Consequently, *memory is often the limiting resource* in the consolidation process [5]. Several solutions tried to address this problem by minimizing VMs memory footprints (e.g. page sharing [6, 7, 8], page compression [9, 10]) and allowing memory overcommitment (ballooning [11, 12, 7]). However, these techniques are generally not employed in mainstream cloud DCs for several reasons. First, despite the achieved memory gains, they all have a detrimental impact on the performance of user applications. Second and specifically for ballooning, these solutions depend on efficiently estimating the memory footprint which is a fairly difficult task considering the variance of the working-set over time [13, 14].

In this paper, we propose a new DC power management system called *Drowsy-DC*, based on an innovative idea which combines concepts from the two previous categories: we state that using consolidation, we can create situations where a DC server may be *suspended despite not being empty* (i.e. it is hosting VMs), thus greatly improving its power consumption.

We observed patterns in VMs activity traces. From the point of view of their activity patterns, VMs may be classified in three categories: short-lived mostly-used VMs (noted SLMU, e.g. MapReduce tasks), long-lived mostly-used VMs (noted LLMU, e.g. popular Web services), and long-lived mostly-idle VMs (noted LLMI, e.g. seasonal Web services). In this paper we focus on LLMI VMs, as defined by Zhang et al. [15]. Their activity pattern is composed of isolated operations followed by long periods of idleness. Notice that LLMI VMs do exist in a

[1]The Advanced Configuration and Power Interface (ACPI) is an standard used by OSs to manage the power of hardware components.

proportion significant enough in public clouds, that Amazon and Microsoft recently provided special pricing for such cases, despite the implementation of Function-as-a-Service [16]. As for private clouds host a majority of LLMI VMs: for instance, at least half of Nutanix's workload is made of enterprise applications which are typically LLMI [17]. Based on this observation, the basic idea of Drowsy-DC is to enforce as far as possible[2] the colocation of VMs exhibiting similar *idleness patterns*. In this way, the hosting servers can be suspended during idle periods. Such suspended servers are called *drowsy* servers in the paper. To the best of our knowledge, we are the first to investigate this approach in the context of cloud DCs. To implement it, Drowsy-DC is composed of three main components: a consolidation support module, a suspension subsystem and a resuming subsystem. In summary we make the following contributions:

- A consolidation approach based on modeling VM behavior of idleness, which enables VM colocation based on matching idleness periods.
- An algorithm allowing to efficiently detect the right time for suspending a server. It keeps a server awake as long as any of its VMs need computation power. Our algorithm also prevents the server from quickly alternating between high and low power states.
- An optimized waking system which minimizes performance degradation for interactive workloads.
- The evaluation of each component as well as the global system. Experiments confirm the effectiveness of Drowsy-DC and the energy gains as well as the negligible overhead concerning the consolidation process and the performance of user VMs. Concerning the energy gains, the evaluation results show that Drowsy-DC improves the OpenStack [18] consolidation system (Neat [19]) by about 50%. We also compared Drowsy-DC with another VM consolidation supports named Oasis [20].

The rest of the paper is organized as follows. Section II gives a high-level architecture of a datacenter managed by Drowsy-DC. Sections III to V describe every Drowsy-DC component. We present the evaluation results in section VI. We compare our contributions with related work in section VII. Finally, we conclude in section VIII.

II. General overview

A datacenter managed by Drowsy-DC must include two software modules:

Waking module: an extension to the already existing datacenter manager, which wakes up suspended servers when hosted VMs require computing resources, and includes optimizations to guarantee optimal resuming time;

Suspending module: an extension for server monitoring, that takes the enlightened decision of suspending a server, and works hand-in-hand with the waking module to ensure optimal resuming time.

[2]That is to say, with respect to resource availability.

With these two modules in place, the existing dynamic consolidation solution can be augmented with Drowsy-DC idleness-based consolidation algorithm. The resulting system can make VM placement decisions based on classic criteria, such as resource requirements, as well as the new criterion of VMs' idleness patterns. Drowsy-DC's idleness-based consolidation algorithm is described in the next section.

III. Idleness aware VM placement

The central concept that rules placement decisions in Drowsy-DC is *idle periods*. We want to colocate VMs which are likely to be idle during the next time interval (e.g. the next hour) so that their host can remain suspended during that interval. To this end, Drowsy-DC continually builds each VM's *idleness model* (noted IM), which summarizes its past idleness. By so doing, each time a VM is candidate for placement, Drowsy-DC derives from its IM an *idleness probability* (noted IP), which quantifies the likelihood of this VM being idle in the next time interval. We also define a server's IP as the average of its VMs' IPs: as we add a special consolidation step to keep the range of IPs on a server small (see section III-D), it is better to use the average in order to represent the general behavior of a server from the IPs of its VMs. The placement algorithm then chooses the destination server which satisfies both the traditional placement constraints (e.g. resource availability) to enforce SLA, and the constraint of proximity between the VM's IP and the server's IP while aiming to increase the latter.

This section describes how the IM is built and explains how to compute the IP. Finally, it shows how to integrate into an existing data center management system using OpenStack.

A. Content of the idleness model (IM)

The purpose of a VM's idleness model (IM) is to provide data to compute its idleness probability (IP) for future time intervals. A naive solution is to consider that the IP of the next time interval only depends on the current time interval. However, this leads to a very high false positive rate. Instead, this paper proposes an approach that comes from studying the idleness of VMs from the production DC of Nutanix, a private cloud provider [21]. We identified three types of VMs: short-lived mostly-used VMs (noted SLMU, e.g. MapReduce tasks), long-lived mostly-used VMs (noted LLMU, e.g. popular web services), and long-lived mostly-idle VMs (noted LLMI, e.g. seasonal web services). This paper focuses on LLMI VMs. We extensively inquired about them and we found a periodic idleness at four different scales: (1) the hour in the day (e.g. morning); (2) the day in the week (e.g. week-end); (3) the day in the month (e.g. end of the month); and (4) the month in the year (e.g. seasons). This observation is in line with other trace analysis works [4, 22, 23].

Based on this study, we decided to use the hour as the time interval, which can be seen as the resolution of the IM — and thus the resolution of the IP-based placement decisions. Nonetheless, in order to define a VM's idleness, we take into account all four scales presented above. We want to express the following information: "the probability the VM is idle at hour h, on the day d_w of the week, which is also the day d_m of the month m, is X". For instance, a national diploma results website is mostly used at some specific hours (2 p.m., 3 p.m.) of a specific day (20th) of one month (July), every year.

Each server runs a model builder which collects every hour the activity level of each VM and updates its *synthesized idleness* (SI) scores contained in its IM (see section III-C). The activity level of a VM is based on the number of scheduler quanta that were allocated to the VM during an hour. There are four types of SI scores, for each time scale:

- $SI_d(h)$: synthesized idleness during h regarding its position in the day;
- $SI_w(h, d_w)$: synthesized idleness depending on h and the day of the week d_w;
- $SI_m(h, d_m)$: synthesized idleness depending on h and the day of the month d_m;
- $SI_y(h, d_m, m)$: synthesized idleness depending on h, d_m and the month m of the year.

The model also takes into account the importance of the time scale (represented by each type of SI scores) in the VM's idleness. For instance, in the previous diploma results website example, we can see that the position of the hour in the week (SI_w) is the least important factor in the idleness periodicity. Therefore, each time scale is given a *weight* — higher means more important. Its value is periodically corrected (see section III-C for the update algorithm).

In summary, a VM's idleness model is composed of many synthesized idleness scores ($24\ SI_d, 24 \times 7\ SI_w, 24 \times 31\ SI_m, 24 \times 365\ SI_y$) and 4 weights ($w_d$, w_w, w_m, and w_y).

B. Computing the idleness probability (IP)

Having the VM's model, its *idleness probability* (IP) for a given hour h of the day d_w of the week, which is also the day d_m of the month m, is computed using the formula in eq. (1).

$$\begin{aligned}\mathrm{IP}(h, d_w, d_m, m) &= w_d \cdot SI_d(h) + w_w \cdot SI_w(h, d_w)\\ &\quad + w_m \cdot SI_m(h, d_m) + w_y \cdot SI_y(h, d_m, m)\\ &= \mathbf{w}^\intercal \cdot \mathbf{SI}\end{aligned} \tag{1}$$

$\mathbf{w}$ is the vector of the weights ($^\intercal$ is the transpose operator), and $\mathbf{SI}$ is the vector of the four SI scores associated with the time interval for which the IP is being computed.

C. Updating the idleness model

As said above, a VM's IM is revised each hour: its SI scores are updated and the weights are corrected.

a) Computing SI_* scores: At VM creation time, all SI_* are set to zero (i.e. undetermined behavior); they will be kept in bounds $[-1, 1]$ when being updated. Further, they are updated periodically at the end of each hour in the following way: if the VM was seen idle the whole hour, SI_* are incremented, otherwise they are decremented. The update value $v(SI_*)$ that is added to or removed from a SI_* is calculated as follows.

The update value depends first on the activity level a of the VM: either the activity level a_h of the hour considered for update if the VM was active, or the average activity level $\overline{a}$ of past active hours if the VM was idle. This way, whenever a VM is seen idle during an hour after showing high activity levels during active hours, its SI_* for this hour increases fast to indicate that seeing idleness is significant. The activity level is the ratio of CPU quanta scheduled for the VM, over the total possible quanta during an hour; very short scheduling quanta — noise — are filtered out. Choosing the activity level for the update value v is summarized in eq. (2).

$$a = \begin{cases} a_h, & \text{if } a_h > 0. \\ \overline{a}, & \text{if } a_h = 0. \end{cases} \tag{2}$$

The activity value is then scaled to the SI_* bounds $[-1, 1]$ to give a^* (eq. (3)). The scaling ratio $\sigma = \frac{1}{365 \times 24}$ is defined so that a VM needs constant activity ($a_h = 1$) during an entire year to bring its SI_d from 0 to -1 (ignoring the coefficient u described below).

$$a^* = \sigma \cdot a = \frac{a}{365 \times 24} \tag{3}$$

Then, the update value v also depends on the current value of the SI_* via the coefficient $u(|SI_*|)$ expressed in eq. (4) (notice that we use the absolute value of SI_*). It exists so that (1) SI_* increase or decrease quickly when undetermined to learn the VM's behavior quickly; and (2) SI_* do not reach very extreme values so that the IM can respond to unexpected VM behavior quickly. In eq. (4), α and β are used to control the effect of u: α can be seen as the decrease speed of the update value when the threshold set by β is reached; and β is interpreted as the threshold above which the SI_* (in absolute value) is considered to start reaching extreme values. α was empirically set to 0.7, while β was set to 0.5 (halfway between undetermined and determined). We did not explore the possibility of dynamically setting α nor β based on VM activity level variations, which could be a way for improvement.

$$u(|SI_*|) = \frac{1}{1 + \mathrm{e}^{\alpha(|SI_*| - \beta)}} \tag{4}$$

Finally, the update value $v(SI_*)$ that is added to (if $a_h = 0$) or removed from (if $a_h > 0$) SI_* is:

$$v(SI_*) = a^* \cdot u(|SI_*|) \tag{5}$$

b) Computing weights: The weights are learned throughout the lifetime of the VM: they are recomputed and corrected after each hour. To this end, we use an unsupervised feature learning method which consists in minimizing the quadratic error function Q defined in eq. (6).

$$Q(\mathbf{w}) = (\mathrm{IP}' - \mathrm{IP})^2 \tag{6}$$

IP' is the IP that should have been predicted given full knowledge, and IP is the IP that is calculated with the weights that are under learning process. However, due to its nature of *probability estimation*, there is no correct value for IP'. Thus it is replaced with the mixed expression given in eq. (7).

$$\mathrm{IP}' = \mathbf{w_0}^\intercal \cdot \mathbf{SI}' \tag{7}$$

$\mathbf{w_0}^\intercal$ is the transpose of the weight vector at the beginning of h and $\mathbf{SI}'$ is the vector of the four SI scores with their updated values. Therefore, the expression of the quadratic error function that we seek to minimize in order to learn the weights becomes as shown in eq. (8).

$$Q(\mathbf{w}) = (\mathbf{w_0}^\intercal \cdot \mathbf{SI}' - \mathbf{w}^\intercal \cdot \mathbf{SI})^2 \tag{8}$$

In order to minimize this function, we use a steepest descent algorithm [24]. It iteratively takes steps proportional to the negative of Q's gradient, thus converging toward the value of $\mathbf{w}$ that minimizes the quadratic error. It has the advantage of being fast, while yielding good results. Its precision can be set to not incur any overhead in the consolidation system.

D. Integration with OpenStack

Our contribution applies to any cloud management system. We use OpenStack [18] for illustration. Before presenting how our VM placement algorithm can be integrated into OpenStack, let us first introduce its VM placement-related components.

A VM placement operation can be triggered for two reasons: VM creation and dynamic VM consolidation. In OpenStack, the former is handled by the cluster manager (Nova). It includes a Filter Scheduler which selects suitable hosts for the VM, by executing the following steps: (1) discard the unsuitable hosts based on a large panel of parameters such as available resources; and (2) weight and sort the remaining hosts based on parameters like colocation rating. Concerning VM consolidation, OpenStack relies on Neat which splits the problem into four sub-problems [25]: (1) determine the underloaded hosts (all their VMs should be migrated and the hosts should be switched to low-power state); (2) determine the overloaded hosts (some of their VMs should be migrated in order to meet the QoS requirements); (3) select VMs to migrate; and (4) place the selected VMs to other hosts. Because Nova and Neat are flexible, we can easily implement our idleness-aware placement algorithm.

a) Initial placement at VM creation time: Nova allows an easy integration of new filters and weighers. In order to integrate our solution, we added our own weigher so as to favor hosts with best-matching idleness probability.

b) VM migration at consolidation time: Neat is designed such that one can plug in a custom consolidation algorithm. Concerning the algorithm presented above, we are interested in steps (3) and (4). We have adjusted them as follows:

(3) Besides the classic parameters involved in selecting the VMs to migrate (e.g. migration speed), we select the ones with the IP the furthest from the host's IP. We sort VMs first by classic criteria, and then by decreasing distance between their IP and their host's IP. Thus VMs with the most different IPs are selected first, and then for a similar distance between IPs[3] the classic criteria are used.

(4) For a VM to migrate, we want to select the destination host with the closest IP. We first treat VMs with the biggest resource requirements, and then find the hosts that can host it; among all the suitable hosts, we then select the one with the IP the closest to the VM's IP.

After Neat has managed overloaded and underloaded host, the VMs in the DC occupy a minimal set of hosts. Drowsy-DC adds an opportunistic consolidation step that is purely based on the IP. Imitating Neat's process, it does the following:

(1) Determine the hosts with a range of VMs' IP that is too wide: on a same host, if the IP of the most active VM (lowest IP) and the IP of the most idle VM (highest IP) are too far apart, Drowsy-DC must migrate the VMs with the most extreme values of IP until the IP range is under a threshold. We empirically set the threshold of a too wide IP range to 7σ (with σ the activity scaling factor defined in section III-C): it roughly represents a difference of a week of constant maximum activity in a SI_d.
(2) Select VMs to migrate: they are the VMs with the IP the most different with the host's IP (recall that the host's IP is the average of its VMs' IPs).
(3) Place the selected VMs to other hosts: this is the same algorithm as when treating overloaded or underloaded hosts (see hereabove).

The overall goal of IP-augmented consolidation is *to put VMs with similar IPs together*. The rationale is that servers with VMs of high IPs have a high chance of sleeping, while servers with VMs of low IPs will probably never sleep. The latter are servers where Drowsy-DC can do nothing, because they do not host LLMI VMs. Among them, normal performance issues are addressed by the classic consolidation algorithm. Servers with VMs of average IPs — thus VMs of undetermined nature — are separated from servers with VMs of high IPs, but still have a better chance to go to sleep than servers with VMs of low IPs. They also serve as initial hosts for newly scheduled VMs, until the nature — LLMI or not — of these VMs is learned.

c) Synthesis: As a final note, we stress the fact that there is *no overhead* in the case of wrong predictions. As explained above, the IM is solely used by the consolidation algorithm for hinting at VM placement. Actual suspension or wake up of a server is always executed because of real factors such as host activity or incoming query, by the suspending or the waking modules that are described in the following sections.

IV. Host suspension

As shown in section II, Drowsy-DC adds a suspending module to managed hosts. This software addition monitors its host's idleness and takes the decision of suspending it. It also communicates crucial information to the waking module — including a waking date — as explained in section V-B. We detail here the factors that the suspending module takes into account before suspending its host.

In a naive way, a system is idle if none of its processes is in the running state. However, there are *false negatives and false positives.* The former are processes that are running but

[3]There is a tolerance when sorting by distance between VM's IP and host's IP so close distances are considered equal.

should not be considered as such in this context. This includes monitoring solutions running on the drowsy server, or kernel-related background services such as watchdogs. We easily address this issue with a black-listing system.

False positives are VMs' processes that are not running but the service they provide must not be considered idle. First, a process may be blocked waiting for resources, such as a disk read: in this case, the drowsy server should not be suspended. This highlights the need to determine the *reason* a process is not running. Second, a VM's process may be idle but the service embedded in it may not. For instance, the service may have open sessions or connections such as SSH or TCP. If nothing is exchanged on them, the VM is seen idle but suspending the drowsy server would induce an unexpected latency. Identifying this second type of false positives mostly requires some kind of introspection [26] to get parameters that are linked to the real activity of the VM. We do not adopt this way for two reasons. First we want to implement a system which is able to work with unmodified applications. Second, we mitigate the potential overhead by implementing a very quick resuming mechanism (see section V). It is also possible to use a heuristic based on the fraction of currently used resources. One example of a metric is VM page dirtying rate, that can be monitored from the hypervisor [20].

Moreover, the idleness monitoring takes into account a *grace time:* when a drowsy server is resumed, there is some time during which it cannot be suspended again, whatever its activity level. This is to avoid an oscillation effect of servers alternating between fully awake and suspended states, which would incur unwanted behavior, bad quality of service and increased power consumption. The grace time is calculated by the suspending module when the host resumes, based on the idleness probability of the host: if the IP tells that it is likely that the host is active, the grace time is longer to accommodate for predicted activity and avoid overhead. We empirically set the grace time between 5s and 2min, exponentially increasing as the IP decreases in order to be conservative with the quality of service of undetermined and active VMs.

V. Host waking

Guaranteeing the quick waking of a drowsy server is an essential part of Drowsy-DC. This is under the responsibility of the *waking module,* located on a server that manages the datacenter, and for this purpose never sleeps. For scalability purposes, one waking module can be used per rack, instead of one component for the entire DC. In our prototype, it is located on the software defined network (SDN) switch. Moreover, knowing that the waking module is at the heart of our solution, its implementation is fault tolerant. To this end, all waking modules work in a collaborated manner. Each waking module monitors — via a heart beat mechanism — and mirrors another one. In this way, when a waking module is defective, it is replaced with an identical version.

Two event types can trigger a server resume: (1) inbound network request; and (2) scheduled waking date.

A. Waking on an inbound network request

The waking module includes a lightweight packet analyzer. Each request received by the SDN switch is first analyzed in order to check whether the destination VM is hosted on one of the currently suspended servers. This is performed efficiently thanks to a hashmap, mapping VMs IP addresses to the MAC addresses of the drowsy servers that host them.[4] If the destination server is indeed suspended, the waking module sends it a Wake-on-LAN (WoL) packet beforehand.

B. Waking on a scheduled date

Upon suspending its host, the suspending module described in section IV computes a *waking date*. To this end, it scans the high-resolution timers[5] that are registered in the kernel. When a process sleeps, it registers a timer which will wake it up when the time comes. The waking date is then the earliest of these high-resolution timers. In practice, we obtain this information via a helper kernel module we developed, that walks the red-black tree structure that is used internally by the kernel to store the timers. This may yield false positives, i.e. timers of processes that shouldn't trigger the waking of the host. They are most likely the same processes as the false negatives that the suspending module ignores when checking the host's idleness (see section IV). Thus, we filter the timers according to the processes that registered them.

Because of the filtering, it may happen that no timer is valid when choosing a waking date: it means that no work of interest is scheduled. The host can remain suspended indefinitely until the waking module wakes it up because of an external request.

Finally, before suspending its host, the suspending module sends the waking module the scheduled waking date. The waking module manages a hashmap, that maps waking dates to the MAC addresses of the hosts that registered them. Subsequently, when a waking date approaches, the waking module sends a WoL packet to the associated drowsy server (and removes the mapping). This request is sent ahead of time in order to take into account the waking latency.

VI. Evaluation

A. Evaluation in a real environment

1) Methodology: This experiment demonstrates the effectiveness of Drowsy-DC: we make it periodically relocate all VMs, instead of waiting for the need of a migration decision (e.g. when a host is overloaded). This behavior is unpractical because it would perform too many migrations in a real situation — leading to performance degradation across the datacenter, but this allows to observe the efficacy of Drowsy-DC. Moreover, while Drowsy-DC is always evaluated with the suspended power state enabled on the hosts, we compare it to Neat with both suspension disabled (current real world case) and with suspension enabled. Transitioning to suspended state is based on the exact same algorithm as Drowsy-DC, the grace time

[4]The VM to host mappings are only updated when a host is suspended.

[5]These timers are designated as "high-resolution" in the Linux kernel because they usually feature a resolution of a few nanoseconds.

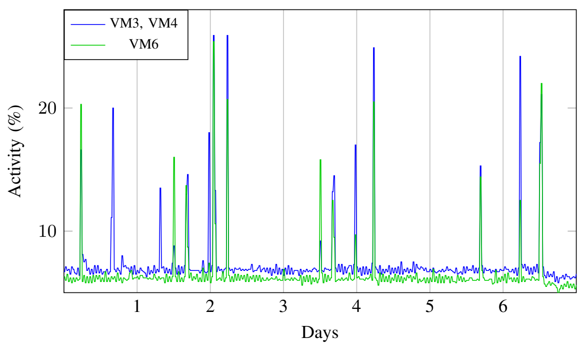

Figure 1. Examples of real workloads we used.

excepted because it requires computing idleness models, which is a Drowsy-DC feature (see section IV). Comparing Drowsy-DC to Neat with suspension enabled shows the usefulness of our idleness probability-based consolidation.

2) Experimental setup: To this end, we built an OpenStack cluster composed of six HP machines (noted P_1-P_6) embedding Intel Core i7-3770 CPU @ 3.40GHz processors, 16GB memory, 10GB network cards, running Ubuntu Server 14.04, and virtualized with QEMU/KVM. They are linked together with a software defined network (SDN) switch, provided by P_1. This one also hosts both the waking module and all the OpenStack controllers. OpenStack uses P_2-P_5 as the resource pool. The cluster hosts 8 VMs (6GB memory and 2 vCPUs each, maximum 2 VMs per machine) set up as follows: 2 LLMU VMs (noted V_1 and V_2) and 6 LLMI VMs (noted V_3-V_8). Each VM runs an application from CloudSuite [27]: Media Streaming for LLMU VMs and Web Search for LLMI VMs. P_6 hosts all CloudSuite client simulators. Web Search client simulators are configured to generate the traces of 5 VMs we monitored during seven days in Nutanix's private production DC, with V_3 and V_4 receiving the exact same workload. Figure 1 depicts a few of these traces. The two LLMU VMs are initially placed on distinct machines. Every machine implements the ACPI S3 state (suspend to RAM). The energy consumed by a host when suspended is about 5W, around 10% of the consumption in idle S_0 state. The evaluation results are as follows.

3) General results: Figure 2 shows the percentage of time each VM co-ran with every other VM. We can see that Drowsy-DC accurately identified that V_1 and V_2 are LLMU VMs, thus they were packed on the same machine for the majority of the experiment. It also predicted among LLMI VMs those having the same idleness periods in the near future, and packed them on the same machine during the matching periods. For instance V_3 and V_4, which received the same workload (see fig. 1), also shared the same machine for a significant duration and after only one migration of V_4. The last column of fig. 2 shows the number of migrations each VM experienced: it is low, meaning that a migrated VM reaches a stable state.

Moreover, we measured the fraction of time each machine spent in suspended state,[6] with Drowsy-DC's IP-based consolidation and with Neat; results are shown in table I. In total,

[6]Because there were 8 VMs and 4 hosts that could only host exactly 2 VMs each, no host ever slept, i.e. ever transitioned to "suspend to disk" state.

	V_1	V_2	V_3	V_4	V_5	V_6	V_7	V_8	#mig
V_1	100	85	0	0	0	15	0	0	1
V_2	85	100	0	0	0	0	0	15	0
V_3	0	0	100	76	0	0	24	0	0
V_4	0	0	76	100	23	0	0	1	1
V_5	0	0	0	23	100	77	0	0	2
V_6	15	0	0	0	77	100	0	8	1
V_7	0	0	24	0	0	0	100	76	1
V_8	0	15	0	1	0	8	76	100	3

Figure 2. Colocation percentage of each VM — black cells: V_1 and V_2 were LLMU VMs; dark gray cells: V_3 and V_4 received the same workload. Last column is the number of migrations a VM experienced.

Table I
FRACTION OF TIME (PERCENT) SPENT BY HOSTS IN SUSPENDED POWER STATE, WITH DROWSY-DC AND WITH NEAT

Algorithm	P_2	P_3	P_4	P_5	Global
Drowsy-DC	0	94	79	91	66
Neat	89	7	8	93	49

hosts managed by Drowsy-DC were suspended for a duration 35% longer than with Neat; i.e. Drowsy-DC's consolidation algorithm that optimizes VM placement in order to maximize periods of host suspension, increased the total duration of such periods by 35%. Notice that in Drowsy-DC's evaluation, P_2 is the machine which eventually hosted the two LLMU VMs (V_2 was initially placed on it), so it was never suspended.

In summary, Drowsy-DC reduced the total energy consumption by about 55%, 18kWh instead of 40kWh when consolidating using Neat, with host suspension disabled. Evaluation with Neat and enabled suspension shows a consumption of 24kWh, which means that Drowsy-DC's consolidation algorithm saved 27% of energy compared with simply implementing the S3 power state. As a side note, Drowsy-DC's effectiveness increases with time, as idleness models get updated and as the consolidation algorithm continues to make better placement decisions than Neat idleness-wise.

Last but not least, we observed that Drowsy-DC guarantees all application's SLA. For instance, more than 99% of the web search requests were serviced within 200ms as required by CloudSuite. However, we observed that the response time of every request triggering the waking of a drowsy server was up to about 1500ms. This does not impact the overall SLA because such requests are a minority, and our work on quick resume brings down the waking time to 800ms.

As a final note, we also experimented Drowsy-DC with applications that rely on timers for triggering their activity (a backup service in our case). We observed the effectiveness of every Drowsy-DC's module, while incurring no performance degradation. The latter is explained by the fact that the waking module anticipates the timer expiration date — which is provided in advance by the suspending module, thus it wakes up the drowsy server ahead of time.

4) Specific results: the suspending module: We evaluated this module from three perspectives: (1) effectiveness (detection of idle states, prevention of power states oscillations and calculation of the next working date); (2) overhead (resource consumption and suspension time); and (3) scalability (evolution of overhead when increasing the number of VMs).[7] The evaluation results are the following.

The effectiveness of the suspending module concerning detection of idle states and calculation of the next working date is close to perfect as it relies on the appropriate Linux and KVM tools. The grace time helps to prevent about 63% percent of the quick power states oscillations (a host suspending and resuming in less than 2min) while wasting only 9% of potential suspension time. The time taken by a transition to suspended state is about the same (around 1s) as in vanilla Linux. This time is constant regardless of the number of VMs because transitioning to S3 requires no checkpointing to disk which could have been the origin of a scalability issue [28]. Finally, the suspending module consumes less than 0.1% of the server's resources (CPU and memory) whatever the number of VMs, as it performs very simple tasks.

5) Specific results: the waking module: The waking module runs alongside the SDN switch in the rack. It has been evaluated from the same perspectives as the suspending module, augmented with the resuming time of a drowsy server.

About its capability to detect that a drowsy server must be woken up, we observed perfect results as this feature is based on exact information: host state and VM placement are sent by the suspending module to the waking module, and the VM targeted by an external query is determined by looking at the network datagram to get the destination address. About the overhead and the scalability, we realized several experiments while varying the traffic volume handled by the waking module: from 1GBps up to 10GBps (the SDN switch capacity). Figure 3 presents the results of these experiments. We can observe from fig. 3 top a constant memory need (about 10%) while the CPU consumption increases (from 8% up to 16%).

Concerning the resuming time, it is composed of three sub-values: (1) detection time (time taken by the waking module to determine the target drowsy server); (2) network time (time taken by the WoL magic packet to reach the drowsy server); and (3) drowsy server resuming time. We can see from fig. 3 bottom that both the detection and resuming times are almost constant (respectively about 50ms and 2.3s). The network time increases with the network traffic (from 0.05ms up to 0.25ms). This is not an issue because the network time is not the dominant one: the server resuming time is. The next section describes how we optimized the resuming process to shorten it.

6) Optimization of server resuming: The server resuming time depends on three parameters: (1) firmware resuming (484ms); (2) kernel resuming (295ms); and (3) network card reset (1.5s). The first is independent of the kernel, and hardware constructors' participation is needed to shorten it. We worked on improving the two others.

a) Kernel resuming: The duration of this phase is broken down in three sub-phases: 26ms to resume CPUs, 276ms to resume drivers, and 3ms to resume user space processes. We observed that the first two sub-phases are the longest, and optimized them as follows.

[7]Up to 32 VMs, the maximum number of instances that an EC2 dedicated host can run at the same time.

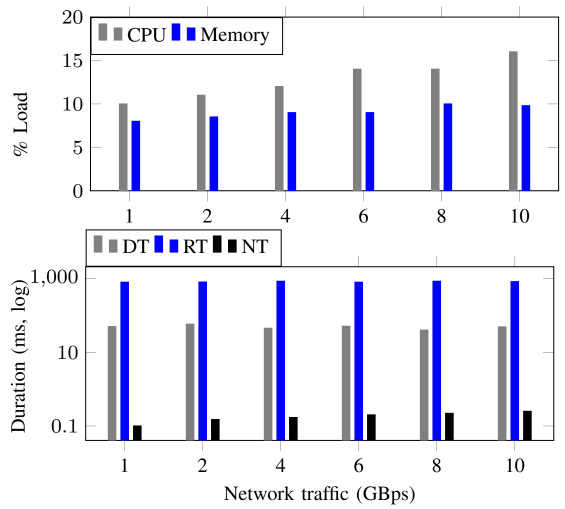

Figure 3. Waking module resource usage (top) and drowsy server waking time (bottom) — DT: detection time; RT: resuming time; NT: network time.

For the first one, CPU resuming, we used Intel's Analyze-Suspend tool[8] to assert that CPUs are resumed sequentially. The optimization consists in resuming only the subset of CPUs that is necessary to handle the inbound query.

As for the second one, driver resuming, it represents the major part of kernel resuming. It is already parallelized, however some modules can be disabled as they manage unused hardware. For instance, the i915 module is bound to the VGA chip, which is mostly optional for a server usage. Disabling all such unused modules reduces the kernel resuming time to a minimum: the time taken by the slowest necessary driver module (the network card driver on our machine). Actual gain may vary following which modules can be disabled. In particular, we selectively choose drivers and devices to resume based on which VM requires the drowsy server to wake up: block storages, network adapters, etc.

b) Network card reset: This phase takes the longest part of the server resuming time (about 1.5s) because transitioning the server to S3 *resets the network card* (NIC). To mitigate this issue, Intel built network cards embedding chipsets (such as the I350 [29]) which are able to keep the physical link up while the system is suspended. This feature is called *critical session* (noted CS) [30].[9] The energy impact of keeping the link up is negligible: the I350 model consumes less than 2mW.

Putting these optimizations altogether, the *total resuming time* is expected to fall under 800ms.

7) Specific results: The idleness model (IM): This section evaluates the IM's capacity to predict idleness periods. We built the idleness models of several idleness patterns described in table II over three years (first column refers to sub-figures in fig. 4). Notice that except for the real traces, idleness patterns are crafted to match the ones observed in traces as described in section III, but with a better resolution of one minute.

[8]https://01.org/suspendresume

[9]It is also known as the veto bit, the "keep PHY link up" capability, and manageability session.

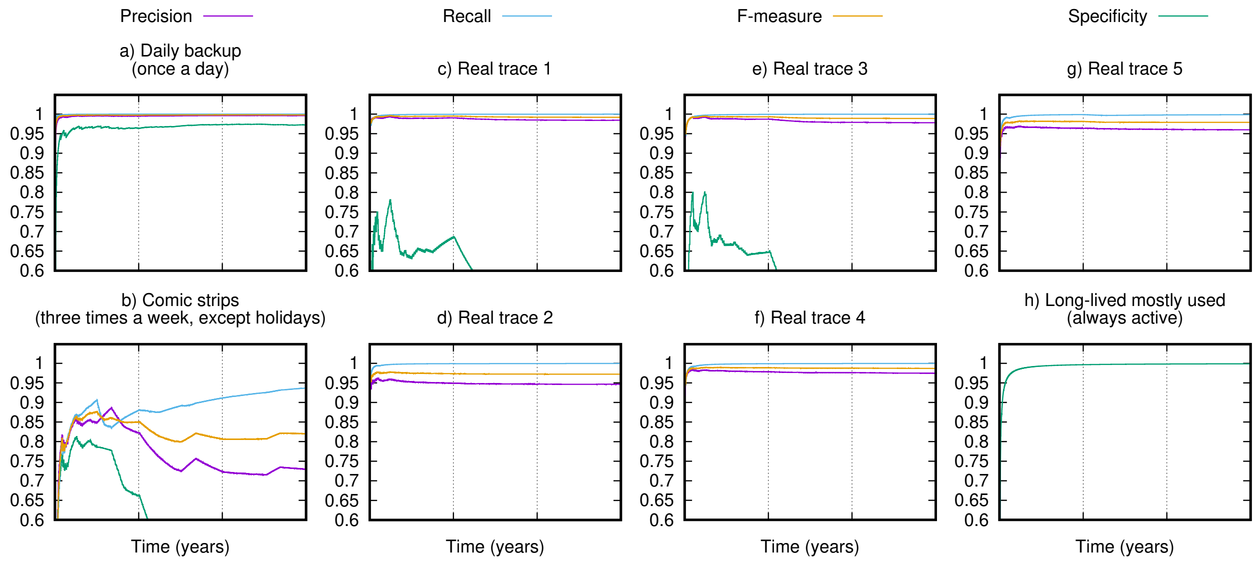

Figure 4. Idleness model efficiency: evaluation of idleness modeling over 3 years (higher is better). Except for subfig. h whose evaluation metric is specificity, the reader should focus on F-measure.

Table II
TRACE TYPES FOR IDLENESS MODEL EVALUATION

Subfig.	Periodicity	Description
a	daily	backup service running each day at 2am
b	three times a week, yearly	online comic strip publication, none in July nor August
c~g	daily, weekly	real traces from production DC (see fig. 1), extended from one week to three years
h	none	long-lived, mostly used VM

Table III
EFFICIENCY METRICS FOR IM EVALUATION

Recall	Precision	F-measure	Specificity
$\frac{TP}{TP+FN}$	$\frac{TP}{TP+FP}$	$\frac{2\times\text{recall}\times\text{precision}}{\text{recall}+\text{precision}}$	$\frac{TN}{TN+FP}$

The IM's objective is to predict whether the VM will be idle during the next hour, so we use four standard prediction accuracy metrics shown in table III to evaluate its efficiency. In the table, TP is the number of true positives; FP is the number of false positives; and same with N for negatives. The case is positive when the VM is idle, or predicted idle — its IP is higher than 50%.

Recall is sensitive to false negative cases, that is to say cases where the model predicted activity but the VM was actually idle; while Precision is sensitive to false positives, cases where the VM was predicted idle but was actually active. We also add Specificity, which is the equivalent of Precision for negative cases: it characterizes the capacity of the model to predict active periods of the VM, and is important for LLMU VMs. Finally, the F-measure summarizes both Recall and Precision, and is the main evaluation score. However, avoiding false positives is especially important: predicting idleness for an active VM, could have it colocated with idle VMs, preventing their host to be suspended and loosing power saving opportunities. Thus Precision is also an important metric on its own. Figure 4 presents the evaluation results. Evaluation was done over three years to show a more complex pattern such as in subfig. b.

a) LLMI VM results: (figures 4 (a)–(g)) First the model needs some time to gain in accuracy — there is a short ramp-up at the beginning of each curve, because it is an unsupervised learning technique. For some VM traces, this first knowledge is enough for their lifetime (e.g. real traces, subfig. c to g); in more complex cases, the IM needs improvement over months. This is the case for subfig. b, which requires about 2 years to completely learn the periodicity: we observe a change in prediction quality when learning the idleness during the holidays months. The beginning of the second year also shows a loss of precision, because it takes some time for the IM to understand that the day of the year has no influence during this period; the beginning of the third year is more stable because the IM now knows it. For the predictable case of subfig. a, and for the real traces, the IM provides very good prediction results, with an F-measure of more than 97% after a few weeks. Even for the more complex case of subfig. b, the F-measure is about 82%.

b) LLMU and SLMU VM results: (figure 4 (h)) We also evaluated the model with a mostly used VM trace. We can see that the model perfectly and quickly recognizes such workloads (Specificity is very close to 1) since they are almost constantly active.

B. Evaluation with simulations

This section presents the evaluation results of Drowsy-DC simulated with real VM traces using CloudSim [31] simulator.[10] LLMU VM traces are provided by Google traces [32] while LLMI VM traces come from the commercial production DC

[10]Drowsy-DC implementation for CloudSim is available here: https://git.bacou.me/?p=Drowsy-DC/SimulationCloudSim.git.

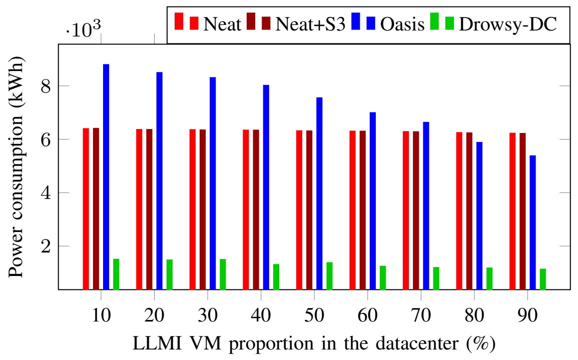

Figure 5. Drowsy-DC comparison with three consolidation methods.

already described in section VI-A. We artificially extended these traces in order to simulate six months of execution. The simulated DC contains 14 servers and hosts 55 VMs. VM and server sizes are randomly generated, using respectively EC2 instance types and EC2 dedicated host types. We compared Drowsy-DC with OpenStack Neat [19], and with Oasis [20]. We experimented several compositions of DC by varying the proportion of LLMI VMs from 10% (e.g. a DC which is massively composed of data processing VMs) to 90% (e.g. a DC which is massively composed of small enterprise Internet services). The results are depicted in fig. 5.

First, we can see that Neat consumes about the same amount of power whatever the fraction of LLMI VMs in the DC. Enabling host suspension offsets a small value to the total consumption (about 10kWh at best). It shows the potential of Drowsy-DC's consolidation algorithm: enabling host suspension with Neat does not lead to significant power savings because Neat does not know how to use this feature to the best. Oasis can actually consume more power because of the memory server added to each host. Its consumption decreases following the increase in fraction of LLMI VMs in the DC, but it only gets compensated with a proportion of LLMI VMs greater than 70%.

Drowsy-DC performs significantly better than the two other methods, with energy savings ranging from 76% (30% of LLMI VMs in the DC) to 81% (70% of LLMI VMs in the DC) compared with Neat. Interestingly, Drowsy-DC is very stable in its power savings across the different fractions of LLMI VMs in the DC, with a standard deviation of 2%. Increasing the proportion of LLMI VMs in the DC does allow Drowsy-DC to consume less power, but this very low standard deviation means that this tendency is very slow. It also means that Drowsy-DC saving the most energy with 70% of LLMI VMs — instead of 90% as intuition would have it – is not significant: difference in power savings between these two proportions is actually less than 0.5kWh. As for Oasis, Drowsy-DC saves 81% power on average. Finally, we observe that with 50% of LLMI VMs in the DC, Drowsy-DC consumes more power than with 40% (again, increase is very small: 5%); it is linked to a great increase in migrations number.

VII. Related work

Several research studies have investigated the problem of power usage in cloud DCs, at many levels.

a) Low-power sleep states (machine level): New paths have been explored about switching compute nodes to a low-power sleep mode. A proposal was SleepServer [33], which addressed the specific issue of desktop company PCs being always powered despite their idle state. Its efficiency was proven [34], but the system relies on proxying the services hosted on the PCs. For cloud DC, some works [28, 3] studied the implementation of low-power idle states into cluster nodes. However, performance degradation is a major issue because of the transition from low-power state to high-performance state. To reduce this overhead, we took inspiration from the latest works from the smartphone world [35] where only the minimal set of hardware is resumed to serve the incoming request, and the device mostly remains suspended.

DreamWeaver [36] is a low-level, architectural support for batch scheduling in order to keep the server in an idle, low-power state as long as possible. It does so by stalling requests up to an acceptable maximum response time. Drowsy-DC is a higher-level solution that relies on VM behavior modelling and consolidation, that does not need any special hardware support (whereas DreamWeaver necessitates a dedicated co-processor).

b) Smart consolidation (datacenter level): Zhi et al. [20] proposed Oasis, which leverages partial-migration of VMs [37]: only the working sets of idle VMs are moved. By doing so, Oasis can densely consolidate the working sets while migrating on-demand the pages accessed by VMs. This allows to put more machines into sleep mode. However the major drawback is that the architecture of Oasis requires additional, low-power memory servers to keep the memory pages available to partially-migrated VMs. An overhead also exists when VMs exit their idle phase and need to be fully-migrated back to their compute machine. Picocenter [15] acts in a similar fashion, as it swaps out idle applications from the cluster, to a cloud storage. Furthermore, it introduces the concept of long-lived mostly-idle applications, which make the scope of this paper. Picocenter however also suffers from a reviving overhead when swapping back in applications, despite a predictive approach to determine the working set for faster resuming.

Meng et al. [38] presented a work which is comparable to Drowsy-DC in the way that the consolidation system should not consider VMs individually, but rather in a joint manner by looking at their activity patterns. However, Meng et al. [38] adopted an opposite strategy: their work packs on the same server VMs which peaks and valleys do not coincide. This way, the unused resources of a low utilized VM can be directed to the other co-located VMs at their peak utilization, thus increasing the consolidation ratio. Concerning Drowsy-DC, its objective is to increase sleeping periods. Thus, it colocates VMs which peaks and valleys coincide. Furthermore, valleys in the case of Meng et al. [38] are not necessarily idle periods as in Drowsy-DC. Finally, both solutions include a prediction algorithm to predict the future state (idle or not) of each VM. However, our algorithm is more general because it is not limited to checking pairs of VMs, and is more scalable (Drowsy-DC's complexity is $\mathcal{O}(n)$, compared to $\mathcal{O}(n^2)$ for the other system [38], with n the number of VMs). This is why Drowsy-DC is suitable for DCs that host an large number of LLMI VMs.

VIII. Conclusion

This paper introduces an innovative management system which aims to reduce the energy consumption in DCs. The system identifies VMs with matching idleness patterns and colocates them on the same physical hosts. The latter are suspended during idle periods, until one of their hosted VMs needs the CPU to accomplish a task. Likewise, our thorough experiments prove its applicability and the benefits for cloud DCs. Depending on the fraction of LLMI VMs in the DC, our system may improve up to 82% upon vanilla OpenStack Neat. Also, our solution outperforms Oasis, a comparable VM consolidation support system, by an average of 81%.

References

[1] L. A. Barroso and U. Hölzle, "The case for energy-proportional computing," *Computer*, no. 12, pp. 33–37, 2007.

[2] C. Delimitrou and C. Kozyrakis, "Quasar: resource-efficient and QoS-aware cluster management," *ACM SIGPLAN Notices*, vol. 49, no. 4, pp. 127–144, 2014.

[3] D. Meisner, B. T. Gold, and T. F. Wenisch, "Powernap: eliminating server idle power," in *ACM SIGPLAN notices*, vol. 44, no. 3. ACM, 2009.

[4] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini, "Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms," in *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 2017, pp. 153–167.

[5] C. Norris, H. Cohen, and B. Cohen, "Leveraging IBM eX5 systems for breakthrough cost and density improvements in virtualized x86 environments," *White paper*, 2011.

[6] S. K. Barker, T. Wood, P. J. Shenoy, and R. K. Sitaraman, "An empirical study of memory sharing in virtual machines." in *USENIX Annual Technical Conference*, 2012, pp. 273–284.

[7] C. A. Waldspurger, "Memory resource management in VMware ESX server," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 181–194, 2002.

[8] G. Miłós, D. G. Murray, S. Hand, and M. A. Fetterman, "Satori: Enlightened page sharing," in *Proceedings of the 2009 conference on USENIX Annual technical conference*, 2009, pp. 1–1.

[9] M. Ekman and P. Stenstrom, "A robust main-memory compression scheme," in *ACM SIGARCH Computer Architecture News*, vol. 33, no. 2. IEEE Computer Society, 2005, pp. 74–85.

[10] J. Zhao, S. Li, J. Chang, J. L. Byrne, L. L. Ramirez, K. Lim, Y. Xie, and P. Faraboschi, "Buri: Scaling big-memory computing with hardware-based memory expansion," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 12, no. 3, p. 31, 2015.

[11] T.-I. Salomie, G. Alonso, T. Roscoe, and K. Elphinstone, "Application level ballooning for efficient server consolidation," in *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM, 2013.

[12] J.-H. Chiang, H.-L. Li, and T.-c. Chiueh, "Working set-based physical memory ballooning." in *ICAC*, 2013, pp. 95–99.

[13] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Geiger: monitoring the buffer cache in a virtual machine environment," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 5, 2006.

[14] P. Lu and K. Shen, "Virtual machine memory access tracing with hypervisor exclusive cache." in *Usenix Annual Technical Conference*, 2007, pp. 29–43.

[15] L. Zhang, J. Litton, F. Cangialosi, T. Benson, D. Levin, and A. Mislove, "Picocenter: Supporting long-lived, mostly-idle applications in cloud environments," in *Proceedings of the Eleventh European Conference on Computer Systems*. ACM, 2016, p. 37.

[16] A. Silver, "Mostly idle at work? Microsoft Azure has some bursty VMs it'd love to sell you," *The Register*, 2017. [Online]. Available: https://www.theregister.co.uk/2017/09/12/microsoft_azure_offers_bursty_vms

[17] "Nutanix investor data sheet," https://s21.q4cdn.com/380967694/files/doc_financials/2019/Q1/Nutanix-Q119-Earnings-Infographics_vFINAL_11272018.pdf, Nutanix, October 2018, online; accessed Dec. 2018.

[18] "Openstack website," https://www.openstack.org/, OpenStack, 2017, online.

[19] A. Beloglazov and R. Buyya, "Openstack Neat: a framework for dynamic and energy-efficient consolidation of virtual machines in OpenStack clouds," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 5, pp. 1310–1333, 2015.

[20] J. Zhi, N. Bila, and E. de Lara, "Oasis: energy proportionality with hybrid server consolidation," in *Proceedings of the Eleventh European Conference on Computer Systems*. ACM, 2016, p. 10.

[21] "Nutanix website," https://www.nutanix.com/, Nutanix, online; accessed Dec. 2018.

[22] O. Beaumont, L. Eyraud-Dubois, and J.-A. Lorenzo-del Castillo, "Analyzing real cluster data for formulating allocation algorithms in cloud platforms," *Parallel Computing*, vol. 54, pp. 83–96, 2016.

[23] G. Amvrosiadis, J. W. Park, G. R. Ganger, G. A. Gibson, E. Baseman, and N. DeBardeleben, "On the diversity of cluster workloads and its impact on research results," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018, pp. 533–546.

[24] D. G. Luenberger, Y. Ye *et al.*, *Linear and nonlinear programming*. Springer, 1984, vol. 2.

[25] 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.

[26] I. K. Kim, S. Zeng, C. Young, J. Hwang, and M. Humphrey, "A supervised learning model for identifying inactive VMs in private cloud data centers," in *Proceedings of the Industrial Track of the 17th International Middleware Conference*. ACM, 2016, p. 2.

[27] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: a study of emerging scale-out workloads on modern hardware," in *ACM SIGPLAN Notices*, vol. 47, no. 4. ACM, 2012, pp. 37–48.

[28] C. Isci, S. McIntosh, J. Kephart, R. Das, J. Hanson, S. Piper, R. Wolford, T. Brey, R. Kantner, A. Ng *et al.*, "Agile, efficient virtualization power management with low-latency server power states," in *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3. ACM, 2013, pp. 96–107.

[29] *Intel Ethernet Controller I350 Datasheet*, http://www.intel.com/content/www/us/en/embedded/products/networking/ethernet-controller-i350-datasheet.html, Intel, 2017, online.

[30] P. Kutch, "Maintaining the Ethernet link to the BMC during server power actions," https://www-ssl.intel.com/content/dam/www/public/us/en/documents/guides/maintaining-the-ethernet-link-to-the-BMC.pdf, Intel, Tech. Rep., 2012, online; accessed Feb. 2017.

[31] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. De Rose, and R. Buyya, "Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Software: Practice and experience*, vol. 41, no. 1, pp. 23–50, 2011.

[32] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving resource efficiency at scale," in *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3. ACM, 2015, pp. 450–462.

[33] Y. A. S. Savage and R. Gupta, "Sleepserver: A software-only approach for reducing the energy consumption of PCs within enterprise environments," *Power (KW)*, vol. 100, no. 150, p. 200, 2010.

[34] J. Reich, M. Goraczko, A. Kansal, and J. Padhye, "Sleepless in Seattle no longer." in *USENIX Annual Technical Conference*, 2010.

[35] M. Lentz, J. Litton, and B. Bhattacharjee, "Drowsy power management," in *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 2015, pp. 230–244.

[36] D. Meisner and T. F. Wenisch, "DreamWeaver: architectural support for deep sleep," in *ACM SIGPLAN Notices*, vol. 47, no. 4. ACM, 2012, pp. 313–324.

[37] N. Bila, E. de Lara, K. Joshi, H. A. Lagar-Cavilla, M. Hiltunen, and M. Satyanarayanan, "Jettison: efficient idle desktop consolidation with partial VM migration," in *Proceedings of the 7th ACM european conference on Computer Systems*. ACM, 2012, pp. 211–224.

[38] X. Meng, C. Isci, J. Kephart, L. Zhang, E. Bouillet, and D. Pendarakis, "Efficient resource provisioning in compute clouds via VM multiplexing," in *Proceedings of the 7th international conference on Autonomic computing*. ACM, 2010, pp. 11–20.