Working set size estimation techniques in virtualized environments: One size does not fit all

Vlad Nitu
Aram Kocharyan*
Hannas Yaya
Alain Tchana
Daniel Hagimont
{first.last}@enseeiht.fr
Toulouse University
Toulouse Institute of Computer Science Research
Toulouse, France

Hrachya Astsatryan
hrach@sci.am
Institute for Informatics and Automation Problem
Yerevan, Armenia

Abstract

Energy consumption is a primary concern for datacenters' management. Numerous datacenters are relying on virtualization, as it provides flexible resource management means such as virtual machine (VM) checkpoint/restart, migration and consolidation. However, one of the main hindrances to server consolidation is physical memory. In nowadays cloud, memory is generally statically allocated to VMs and wasted if not used. Techniques (such as ballooning) were introduced for dynamically reclaiming memory from VMs, such that only the needed memory is provisioned to each VM. However, the challenge is to precisely monitor the needed memory, i.e., the working set of each VM. In this paper, we thoroughly review the main techniques that were proposed for monitoring the working set of VMs. Additionally, we have implemented the main techniques in the Xen hypervisor and we have defined different metrics in order to evaluate their efficiency. Based on the evaluation results, we propose Badis, a system which combines several of the existing solutions, using the right solution at the right time. We also propose a consolidation extension which leverages Badis in order to pack the VMs based on the working set size and not the booked memory. The implementation of all techniques, our proposed system, and the benchmarks we have used are publicly available in order to support further research in this domain.

ACM Reference format:
Vlad Nitu, Aram Kocharyan, Hannas Yaya, Alain Tchana, Daniel Hagimont, and Hrachya Astsatryan. 2018. Working set size estimation techniques in virtualized environments: One size does not fit all. In *Proceedings of International Conference on Measurement and Modeling of Computer Systems, Irvine, California, USA, June 2018 (ACM SIGMETRICS 2018),* 14 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

*Also affiliated with the Institute for Informatics and Automation Problem

1 Introduction

Energy consumption is a primary concern for datacenter (DC) management. Its cost represents a significant part of the total cost of ownership (about 80% [2]) and it is estimated that in 2020, US DCs will spend about $13 billion on energy bills [3].

A majority of DCs implements the Infrastructure as a Service (IaaS) model where *customers* buy (from *providers*) VMs with a set of reserved resources. The VMs host general purpose applications (e.g. web services), as well as High Performance Computing applications. In such IaaS DCs, virtualization is a fundamental technology which allows optimizing the infrastructure by colocating several VMs on the same physical server. Such colocation can be achieved at deployment time by starting as many VMs as possible on each physical machine, or at runtime by dynamically migrating VMs on a reduced set of physical machines, thus implementing a consolidation strategy [4].

Ideally, consolidation should lead to highly loaded servers. Although consolidation may increase server utilization by about 5-10%, it is difficult to actually observe server loads greater than 50% for even the most adapted workloads [5–7]. The main reason is that VM collocation is memory bound, as memory saturates much faster than the CPU. This situation was accentuated over the last several years, as we have seen emerging new applications with growing memory demands, while physical platforms had an opposite tendency; they provide more CPU capacity than physical memory. This mismatch is referred to as *the memory capacity wall* [8].

However, the existing consolidation systems [37, 47] take the CPU as a pivot, i.e. the central element of the consolidation. The memory is considered constant (i.e. the initially booked value) all over the VM's lifetime. Nevertheless, we consider that the memory should be the consolidation pivot since it is the limiting resource. In order to reduce the memory pressure, the consolidation should consider the memory actually consumed (i.e. the VM's working set size) and not the booked memory (see Fig. 1). Thereby, we need mechanisms to (1) evaluate the working set size (WSS) of VMs, (2) to anticipate their memory evolution and (3) to dynamically adjust the VMs' allocated memory. Numerous research papers propose algorithms to estimate the WSS of VMs. However, most of them are able to follow either up-trends (the increase) or down-trends (the decrease) of WSS. The few of them which are able to follow

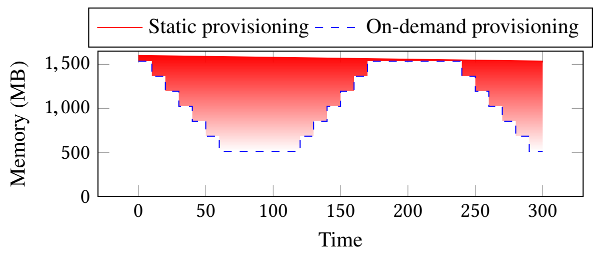

Figure 1. Static provisioning vs on-demand provisioning.

both trends are highly intrusive. Moreover, to the best of our knowledge, no previous work has shown the implications of dynamically adjusting the VM's allocated memory according to the WSS estimation. Finally, as far as we know, no previous consolidation algorithm considers the WSS as a pivot. In this paper we address all the above limitations.

In summary, the contributions of this paper are the following:

- We define evaluation metrics that allow to characterize WSS estimation solutions.
- We evaluate existing WSS techniques on several types of benchmarks. Each solution was implemented in the Xen virtualization system.
- We propose Badis, a WSS monitoring and estimation system which leverages several of the existing solutions in order to provide high estimation accuracy with no codebase intrusiveness. Badis is also able to dynamically adjust the VM's allocated memory based on the WSS estimations.
- We propose a consolidation system extension which leverages Badis for a better consolidation ratio. Both the source and the data sets used for our evaluation are publicly available [1], so that our experiments can be reproduced.

The rest of this article is structured as follows: Section 2 covers a quick background overview. Section 3 presents the general functioning of a WSS estimation solution. Section 4 presents the existing WSS estimation techniques that we analyze and evaluate in this article. Section 5 reports the evaluation results for the main studied techniques. Section 6.1 exposes the details of Badis while Section 6.2 presents the way we integrated Badis in an OpenStack cloud. Section 6.3 evaluates our solution. After a review of related works in Section 7, we present our conclusions in Section 8.

2 Background on virtualization: illustration with Xen

2.1 Generalities

The main goal of virtualization is to multiplex hardware resources between several guest operating systems also called Virtual Machines (VMs). Xen [10] is a well-known virtualization system employed by Amazon [11] to virtualize its DCs. Xen relies on a hypervisor which runs on the bare hardware, and a particular VM (the *dom0*) which includes all OS services. The latter are not included in the hypervisor in order to keep it as lightweight as possible. The other (general purpose) VMs are called *domUs*. In the next subsections, we provide details about memory management and I/O management in Xen, necessary for understanding the WSS techniques we study in this paper.

2.2 Memory and I/O virtualization

In a fully virtualized system, the VM believes it controls the RAM. However, the latter is actually under the control of the hypervisor which ensures its multiplexing between multiple VMs. In this respect, one of the commonly used techniques is the following. The page frame addresses presented to the VM and used in its page tables are fictitious addresses (called pseudo-physical). They do not designate a page frame's actual location in the physical RAM. The real addresses (i.e. host-physical) are known only by the hypervisor which maintains for each *guest page table* in the VMs (mapping guest-virtual → pseudo-physical), an equivalent called *shadow page table* (mapping guest-virtual → host-physical). Each shadow page table is synchronized with its equivalent guest page table. The shadow page tables are the ones used by the MMU[1]. The guest page tables play no role in the address translation process. However, how the hypervisor ensures this synchronization knowing that the VM is a "black box"? In this respect, the hypervisor runs each guest kernel at Ring 3 and sets as read-only the address ranges corresponding to guest page tables. Thereby, any attempt (from the guest kernel) to update a guest page table or the guest %cr3 traps to the hypervisor. Based on the trap error, the hypervisor updates the corresponding shadow page table (in the case of a guest page table write attempt) or switches the execution context (in the case of a guest %cr3 write attempt).

By leveraging this mechanism, a WSS estimation technique can monitor a VM's memory activity in a transparent way, in the hypervisor (see Section 3).

2.3 Ballooning

Memory ballooning [10, 12] is a memory management technique that allows to dynamically reclaim memory from a VM to the hypervisor. Most of the nowadays hypervisors implement this technique in order to reclaim unused memory from VMs, thus avoiding resource waste. In such systems, every VM is equipped with a balloon driver which can be inflated or deflated from the hypervisor/dom0. Fig. 2 presents the general functioning of the balloon driver. Balloon inflation raises memory pressure on the VM, as follows. As soon as the balloon driver receives a higher balloon target size, it allocates a portion of memory and pins it, thus ensuring that memory pages cannot be swapped-out by the VM's OS. Then, the balloon driver reports the addresses of the pinned page to the hypervisor so that it can use them for other purposes (e.g. assigned them to a VM which is lacking memory). In the case of a balloon deflation order, the balloon driver reclaims the pinned pages from the hypervisor and deallocates them. Thereby, the pages reenter under the control of the VM's OS. In Xen, the command *xl mem-set VM_id memory_size* can be used to adjust the balloon target size from the dom0.

[1]The shadow page table's address is loaded into %cr3 at context switch. The CR3 register enables the processor to translate virtual addresses into physical addresses.

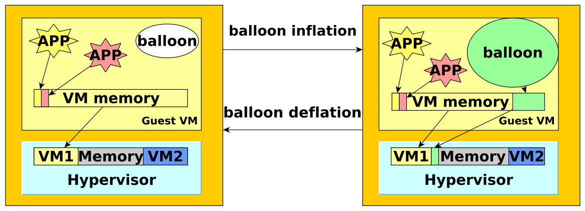

Figure 2. Memory ballooning principles.

3 On-demand memory allocation

3.1 General functioning

As argued in the introduction, the memory is the limiting resource when performing VM collocation. To alleviate this issue, the commonly used approach consists of managing the memory in the same way as the processor, by doing on-demand allocation. Indeed, considering a VM whose booked memory capacity is m_b (representing the SLA that the provider should meet) but which actively uses m_u ($m_u \leq m_b$), the on-demand approach would assign only m_u memory capacity to the VM (instead of m_b as in a static strategy); m_u is called the WSS of the VM. This approach requires the implementation of a feedback loop which operates as follows. The memory activity of each VM is periodically collected and services as the input of a WSS estimation algorithm. Once the latter has estimated the WSS (noted wss_{est}), the VM's memory capacity is adjusted to wss_{est}. In short, the implementation of the on-demand memory allocation strategy raises thee main questions:

- (Q_1) How to obtain the VM's memory activity knowing that the VM is a "black-box" for the cloud provider?
- (Q_2) How to estimate the VM's WSS from the collected data?
- (Q_3) How to update the VM's memory capacity during its execution?

Regarding Q_3, the solution is self-evident. Indeed, it leverages the balloon driver inside the VM (see the previous section). Furthermore, the hypervisor provides an API to control the balloon driver's size. Thus, by inflating or deflating the balloon, the actual memory capacity of the VM can be updated at runtime. The rest of the section focuses on Q_1 and Q_2, which are more complex.

Answering Q_1 raises two challenges. The first one relates to the implementation of the method used for retrieving the memory activity data. The method is either active or passive. An active method modifies the execution of the VM (e.g. deliberately inject page faults) while a passive method does not interfere in the VM's execution process. The active method could impact the VM's performance. For instance, a naive way for capturing all memory accesses may be to invalidate all memory pages in the VM's shadow page table. All subsequent accesses would result in page faults which are trapped by the hypervisor. This solution would be catastrophic for the VM's performance because of the page faults' overhead. The second challenge is related to the level where the method is implemented. Three locations are possible: exclusively inside the hypervisor/dom0, exclusively inside the VM, or spread across both. In the last two locations, the method is said to be *intrusive* because the "black-box" nature of the VM is altered. In this situation, the implementation of the method requires the end-user's agreement. Otherwise, one could exploit only the memory activity data available at the hypervisor/dom0 level. Concerning Q_2, two main challenges should be tackled: the accuracy of the estimation technique (a wrong estimation will either impact the VM's performance or lead to resource waste) and the overhead. In the rest of the paper, the expression "WSS estimation technique" is used to represent a solution to both Q_1 and Q_2.

3.2 Metrics

With respect to the above presentation, the metrics we propose for characterizing a WSS estimation technique are the following: the *intrusiveness* (requires the modification of the VM), the *activeness* (alters the VM's execution flow), the *accuracy*, the overhead on the VM (noted *vm_over*), and the overhead on the hypervisor/dom0 (noted *hyper_over*). Both the *intrusiveness* and the *activeness* are qualitative metrics while the others are quantitative. Among the qualitative metrics, we consider the *intrusiveness* as the most important. We note that the balloon driver alone is not considered an intrusiveness since it is de facto accepted and integrated in most of the OSs. Concerning the quantitative metrics, the ranking is done as follows. Metrics which are related to the VM performance (thus the SLA) occupy higher positions since guaranteeing the SLA is one of the most important provider's objectives. In this respect, we propose the following ranking:

(1) *vm_over*: it directly impacts the VM performance. It could be affected by both the *intrusiveness* and the *activeness*.
(2) *accuracy*: a wrong estimation leads to either performance degradation (under-estimation) or resource waste (over-estimation).
(3) *hyper_over*: a high overhead could saturate the hypervisor/dom0, which are shared components. This could lead, in turn, to the degradation of VMs' performance (e.g. the I/O intensive VMs). In this paper we mainly focus on the CPU load induced by the technique.

The metrics presented above characterize the WSS estimation techniques. Apart from these, we also define a metric which characterizes the WSS itself, namely the *volatility*. The latter represents the degree/speed of WSS variation and is very important for the VM consolidation (see Section 6.2).

4 Studied techniques

This section presents the main WSS estimation techniques proposed by researchers up to the writing time of this paper. We have thoroughly studied them both qualitatively and quantitatively. This section focuses on the former aspect while Section 5 is dedicated to the latter aspect. The presentation of each technique is organized as follows. First, we present the technique description, while highlighting how Q_1 and Q_2 are answered. Second, we explain (whenever necessary) the way in which we implement the technique in Xen (our illustrative virtualization system). Last but not least, we present both the

```
void main(void){
    char* tab=(char*)malloc(2*1024*1024*1024);
    do{
        tab[1]=getchar();
    }while(tab[1]!='a');
    free(tab);
}
```

Figure 3. The `Committed_AS` value increases with the amount of malloc-ed memory even if it is not backed by physical memory.

strengths and the weaknesses of the technique, knowing that they are validated in Section 5.

4.1 Self-ballooning

Description. Self-ballooning [13] entirely relies on the VM, especially the native features of its OS. It considers that the WSS of the VM is given by the `Committed_AS` [14] kernel statistic (`cat /proc/meminfo`), computed as follows. The OS monitors all memory allocation calls (e.g. `malloc`) - Q_1 - and sums up the virtual memory committed to all processes. The OS decrements the *Committed_AS* each time the allocated pages are freed. For illustration, let us consider a process which runs the C program presented in Fig. 3. After the execution of line 2, the value of *Committed_AS* is incremented by 2GB, even if only one octet is actively used. In summary, the *Committed_AS* statistic corresponds to the total number of anonymous memory pages allocated by all processes, but not necessary backed by physical pages.
Implementation. No effort has been required to put in place this technique since it is the default technique already implemented in Xen. The balloon driver (which runs inside the VM) periodically adjusts the allocation size according to the value of the *Committed_AS*.
Comments. As mentioned above, this technique completely depends on the VM. In addition, the implementation of the feedback loop is shift from the hypervisor/dom0 to the VM, making this technique too intrusive. The heuristic used for estimating the WSS is not accurate for two reasons. First, *Committed_AS* does not take into account the page cache, and thus may cause substantial performance degradation for disk I/O intensive applications [15]. Second, this technique could lead to resource waste since the committed memory is most of the times greater than the actively used memory. These two statements are also validated by the evaluation results. The only advantage of the *Committed_AS* technique is its simplicity.

4.2 Zballoond

Description. Zballoond [15] relies on the following observation: when a VM's memory size is larger than or equal to its WSS, the number of swap-in and refault (occurs when a previously evicted page is later accessed) events is close to zero. The basic idea behind *Zballoond* consists in gradually decreasing the VM's memory size until these counters start to become non-zero (the answer of Q_1). Concerning Q_2, the VM's WSS is the lowest memory size which leads the VM to zero swap-in and refault events.
Implementation. *Zballoond* is implemented inside the VM as a kernel module which loops on the following steps. (1) The VM's memory size is initialised to its `Committed_AS` value. (2) Every epoch (e.g. 1 second), the memory is decreased by a percentage of the `Committed_AS` (e.g. 5%). (3) Whenever the `Committed_AS` changes, *Zballoond* considers that the VM's WSS has changed significantly. In this case, the algorithm goes to step (1). Our implementation of *Zballoond* is about 360 LOCs.
Comments. Like the previous technique, *Zballoond* is entirely implemented in the VM's OS. Furthermore, *Zballoond* is very active in the sense that it performs memory pressure on the VM. The overhead introduced by this technique comes from the fact that it actively forces the VM's OS to invoke its page reclamation mechanism (every epoch). Therefore, the overhead depends on both the epoch length and the pressure put on the VM (how much memory to reclaim).

4.3 The VMware technique

Description. The VMware technique [12] is an improvement of the naive method presented in Section 3. Instead of invalidating all memory pages, it relies on a sampling approach which works as follows. Let us note m_{cur} the current VM's memory size. To answer Q_1, the hypervisor periodically and randomly selects n pages from the VM's memory (e.g. $n = 100$) and invalidates them. By so doing, the next access to these pages trap in the hypervisor. The latter counts the number of pages (noted f) among the selected ones which were subject to a non present fault during the previous time interval. The WSS of the VM is $\frac{f}{n} \times m_{cur}$, thus answering Q_2.
Implementation. Two implementations of this technique are possible depending on the way the memory pages are invalidated. A memory page can be invalidated by clearing either the present bit or the accessed bit. In the first implementation the hypervisor counts the number of page faults generated by the selected pages while in the second, it counts the number of pages being accessed (the accessed bit is set) during the previous time frame. Notice that the access bit is automatically set by the hardware each time a page is accessed; no trap is triggered in the hypervisor. The implementation of the two methods requires around 160 LOCs.
Comments. This technique is completely non intrusive. The feedback loop is entirely implemented in the hypervisor/dom0. However, the technique has two main drawbacks. First, the method used for answering Q_1 modifies the execution flow of the VM, which could lead to different performance degradation levels depending on the adopted implementation. The first implementation leads to higher performance degradation comparing to the second implementation. This is explained by the cost of resolving a non-present page fault which is higher than the cost of setting the accessed bit (performed in the hardware). However, the accuracy of the second implementation (the number of accessed pages) could be biased if the hypervisor/dom0 runs another service which clears the accessed bit. Such a situation could occur in a KVM environment because the hypervisor (i.e. Linux) runs services like kswapd (the swap daemon) which monitors and clears the accessed bit. As a

second drawback, this techniques is unable to estimate WSSs greater than the current allocated memory. In the best case, the technique will detect that all monitored pages are accessed, thus estimating the WSS as the current size of the VM.

4.4 Geiger

Description. *Geiger* [16] monitors the evictions and subsequent reloads from the guest OS buffer cache to the swap device (the answer of Q_1). To deal with Q_2, Geiger relies on a technique called the ghost buffer [17]. The latter represents an imaginary memory buffer which extends the VM's physical memory (noted m_{cur}). The size of this buffer (noted m_{ghost}) represents the amount of extra memory which would prevent the VM from swapping-out. Knowing the ghost buffer size, one can compute the VM's WSS using the following formula: $WSS = m_{cur} + m_{ghost}$ if $m_{ghost} > 0$.

Implementation. The first challenge was to isolate the swap traffic from the rest of the disk IO requests. In this respect, we forced the VM to use a different disk backend driver for the swap device (e.g. *xen-blkback*). This driver is patched to implement the *Geiger* monitoring technique as follows. When a page is evicted from the VM's memory, a reference to that page is added to a tail queue in the disk backend driver, located inside the dom0. Later, when a page is read from the swap device, *Geiger* removes its reference from the tail queue and computes the distance D to the head of the queue. D represents the number of extra memory pages needed by the guest OS to prevent the swapping out of that page (i.e. the ghost buffer size at that timestamp). However, to update the VM's memory size after each reloaded page from swap would be too frequent. Thereby, we leverage D values to compute the miss ratio curve [17]. This curve is an array indexed by D which represents how many times we saw the D distance in the last interval. For example, if the computed $D = 50$, we increment `array[50]` by one. When the timer expires, we iterate through the array and we sum up its values until we got X% of its total size. In our implementation, we found out that $X = 95$ yields good results. The index corresponding to the position where the iterator stops represents the number of extra memory pages needed by the VM to preserve 95% of swapped out pages.

Comments. Like the *VMware* technique, *Geiger* is also completely transparent from the VM's point of view. Thereby it does not require the VM user's permission. As stated before, the VM has to be started with a different disk backend driver for the swap device. However, this is not an issue since the VMs are created by the cloud provider so, he is the one deciding the disk backend drivers to be used. Additionally, Geiger has an important drawback which derives from its non-intrusiveness. It is able to estimate the WSS only when the size of the ghost buffer is greater than zero (the VM is in a swapping state). Geiger is inefficient if the VM's WSS is smaller than the current memory allocation.

4.5 Hypervisor Exclusive Cache

Description. The *Exclusive Cache* technique [18] is fairly similar with Geiger in the way that both of them rely on the ghost buffer to estimate the WSS. In the *Exclusive Cache*, each VM has a small amount of memory called direct memory, and the rest of the memory is managed by the hypervisor as an exclusive cache. Once the direct memory is full, the VM will send pages to the hypervisor memory (instead of sending to the swap). Thereby, in the Exclusive Cache technique, the ghost buffer is materialized by a memory buffer managed in the hypervisor.

Implementation. In the same way as Geiger, the Exclusive cache technique is also implemented as an extension to the XEN disk backend driver. In the vanilla driver, the backend receives the pages to be swapped through a shared memory between the VM and dom0. Subsequently, the backend creates a block IO request that is passed further to the block layer. In our implementation, instead of creating the block IO request, we store the VM's page content in a dom0 memory buffer. The latter represents the materialization of the ghost buffer.

Comments. In comparison with Geiger, this technique is more active since it may force the VM in eviction state. However, the performance impact of the *Exclusive cache* technique is lower since the block layer is bypassed and the evicted pages are stored in memory. zo

4.6 Dynamic MPA Ballooning

Description. The Dynamic Memory Pressure Aware (MPA) Ballooning [48] studies the memory management from the perspective of the entire host server. It introduces an additional set of hypercalls through which all VMs report the number of their anonymous pages, file pages and inactive pages to the hypervisor (Q_1). Based on this information, the technique defines three possible memory pressure states: *low* (the sum of anonymous and file pages for all VMs is less than the host's total memory pages), *mild* (the sum of anonymous and file pages is greater than the host's total memory pages) and *heavy* (the sum of anonymous pages is greater than the host's total memory pages); this answers Q_2. Depending on the current memory pressure state, the host server adopts a different memory policy. In the case of low memory pressure, this technique divides the hypervisor's free memory to $nb_{VMs} + 2$ slices. Each slice (called *cushion*) is assigned to a VM as a memory reserve. The two remaining cushions stay in the control of the hypervisor for a sudden memory demand. The cushion may be seen as the exclusive cache in the Hypervisor Exclusive Cache technique. In the mild memory pressure state, the hypervisor reclaims the inactive pages from all VMs and rebalance them in $nb_{VMs} + 1$ cushions. In heavy memory pressure, most of the page cache pages are evicted so the technique rebalance exclusively the anonymous pages.

Comments. This technique has high intrusiveness since it requires additional hypercalls in the guest OS. Thereby, it may be effective in the case of a private data center where the cloud manager has a high degree of control over the guest OS. Additionally, the new hypercalls export precise and important information about the VM's memory layout; this may increase the risk of attacks on VMs.

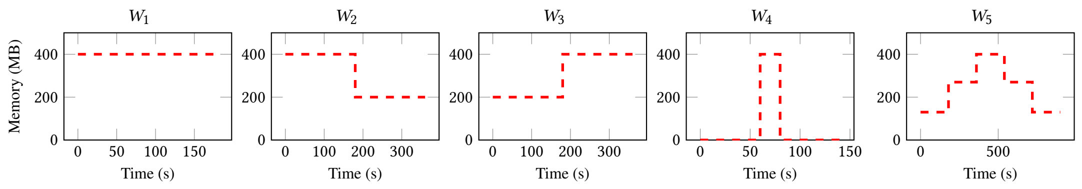

Figure 4. The set of synthetic workloads.

5 Evaluation of the studied techniques

This section presents the evaluation results for most of the techniques described above. We do not evaluate the Dynamic MPA Ballooning since is not a WSS estimation technique. The memory utilization values are directly communicated by the VM to the hypervisor.

5.1 Experimental environment

The experiments were carried out on a 2-socket DELL server. Each socket is composed of 12 Intel Xeon E5-2420 processing units (2.20 GHz), linked to a 8GB NUMA memory node (the machine has a total of 16GB RAM). The virtualization system on the server is Xen 4.2. Both the dom0 and the VMs run Ubuntu server 12.04. One socket of the server is dedicated to dom0 in order to avoid interference with other VMs. Unless otherwise specified each VM is configured with two vCPUs (pined to two processing units) and 2GB memory (the maximum memory it can use).

Concerning the applications which run inside VMs, we rely on both micro and macro benchmarks. The former is an application which performs read and write operations on the entries of an array whose size could be dynamically adjusted in order to mimic a variable workload. Each array entry points to a data structure whose size is equivalent to a memory page. The micro-benchmark allows to compare experimental values with the exact theoretical values, necessary for evaluating the *accuracy* metric. To this end, we build five synthetic workloads which cover the common memory behaviors of a VM during its lifetime. Fig. 4 presents these workloads, noted W_i, $1 \leq i \leq 5$. Each workload is implemented in two ways. In the first implementation (noted $W_{i,s}$), the array size is malloced once, at VM start time, to its maximum possible value. In the second implementation (noted $W_{i,d}$), the array's allocated memory size is adjusted to each step value.

In addition, we also rely on three macro-benchmarks, namely DaCapo [19], CloudSuite [20], and LinkBench [21]. DaCapo is a well known open source java benchmark suite that is widely used by memory management and computer architecture communities [22]. We present the results for 5 DaCapo applications which are the most memory intensive:

- Avrora is a parallel discrete event simulator that performs cycle accurate simulation of a sensor network.
- Batik produces a number of Scalable Vector Graphics (SVG) images based on the unit tests in Apache Batik.
- Eclipse executes some of the (non-gui) jdt performance tests for the Eclipse IDE.
- H2 executes a JDBC-like in-memory benchmark, executing a number of transactions against a model of a banking application.
- Jython intepretes the PyBench python benchmark

CloudSuite is a benchmark suite which covers a broad range of application categories commonly found in today's datacenters. In our experiments, we rely on Data Analytics, a map-reduce application using Mahout (a set of machine learning libraries). LinkBench is a database benchmark developed to evaluate database performance for workloads similar to those at Facebook. The performance metric of all these applications is the complete execution time. By choosing these benchmarks, we wanted to cover the most important and popular applications executed in the cloud nowadays.

5.2 Evaluation with synthetic workloads

As stated above, these evaluations focus on the *accuracy* metric. Fig. 5 and Fig. 6 present the results for each workload and each WSS estimation technique. To facilitate the interpretation of the results, each curve shows both the original workload (noted W_i^o) and the actual estimated WSSs (noted W_{ij}^e), $1 \leq i \leq 5$ (represents the workload type) and j=s,d (represents the implementation type - static or dynamic).

Xen *self-ballooning*, Fig. 5 line 1-2. The accuracy of this technique is very low for all $W_{i,s}$ (see line 1) while it is almost perfect for all $W_{i,d}$ (see line 2). This is because the technique relies on the value of `Committed_AS` as the WSS. Thus, it is able to follow all `Committed_AS` changes. The accuracy of this technique depends on the implementation (i.e. the memory allocation approach) of applications which run inside the VM.

Zballoond, Fig. 5 line 3-4. This technique behaves like *self-ballooning* on all $W_{i,d}$ (see line 4) because it tracks all `Committed_AS` changes. Unlike *self-ballooning*, *Zballoond* is also quite efficient on all $W_{i,s}$ (see line 3). This is because *Zballoond* continuously adjusts the VM's memory size so that swap-in or refault events occur, thus avoiding resource waste. However, if the WSS reduction is faster than the memory reclaim percentage (i.e. 5%), the estimation diverges from the real WSS (see line 3, columns 2 and 4). Even if a higher memory reclaim percentage may solve the problem, this means more memory pressure on the VM and thereby, it would increase the vm_over.

From now on (Fig. 6), we only discuss $W_{i,s}$ results because we observed no difference with $W_{i,d}$ regardless the WSS technique. In fact, only `Committed_AS`-based techniques are sensitive to the way by which the workload is implemented.

VMware, Fig. 6 line 1. Without access to the implementation details of this technique, we considered two versions according to the way the sampled pages are invalidated: the present bit based version (noted $VMware_{present}$) and the access bit based version (noted $VMware_{access}$). The evaluation results of these versions show that they have almost the same accuracy. They are only different from the perspective of other metrics (see the next section). From Fig. 6 line 1, we can see that the *VMware* technique has a main limitation. Although it is able to detect WSS when the VM is wasting memory, it is not able to detect shortage situations. This happens because the percentage of memory pages (among the sampled ones) which is used for estimating the WSS is upper bounded by 100%.

Geiger, Fig. 6 line 2. *Geiger* is the opposite of the *VMware* technique; it is only able to detect shortage situations. This is because it monitors the swap-in and refault events, which only occur when the VM is lacking memory. Another advantage of this technique is its reactivity; it quickly detects WSS changes.

Hypervisor exclusive cache, Fig. 6 line 3. This technique behaves like *Geiger* in the perspective of the *accuracy* metric. They are different in terms of the *vm_over* metric presented in the next section.

5.3 Evaluation with macro-benchmarks

Table 1 presents the evaluation results of each technique with macro-benchmarks. We only focus on the *vm_over* and the *hyper_over* metrics. The *vm_over* value represents the normalized runtime performance of each benchmark while the *hyper_over* represents the normalized CPU utilization by the hypervisor. For example, $vm\_over = 2$ means that the benchmark execution time is twice longer. The interpretation of Table 1 is as follows.

Self-ballooning. It incurs no overhead neither on the hypervisor/dom0 nor on the benchmark.

Zballoond. Like *self-ballooning*, it incurs no overhead on the hypervisor/dom0. However, the VMs' performance is impacted (between 1.09x and 3.67x).

VMware. We can see that the two versions we implemented ($VMware_{present}$ and $VMware_{access}$) incur a relatively low overhead on the hypervisor/dom0. However, the two versions severely impact the benchmark performance (up to 45x degradation in the case of the Data Analytics applications). As presented in the previous section, this is due to the fact that the *VMware* technique is not able to detect memory lacking situations. $VMware_{present}$ leads to more impact on VMs than $VMware_{access}$ (about 3x).

Geiger. Its overhead on either the hypervisor/dom0 or the VM is negligible (less than 2x). Even if the technique does not entirely address the issue of WSS estimation, the VM performance is not strongly impacted since Geiger never leads the VM to a lacking situation like the *VMware* technique.

Exclusive cache. Its overhead on the hypervisor/dom0 is not negligible (about 5x). However, its impact on the VM performance is almost nil (swapped-out pages are store in the main memory).

5.4 Synthesis

Table 2 summarizes the characteristics of each technique according to both qualitative and quantitative criteria presented in Section 3.2. Besides these criteria, the evaluation results reveal that not all solutions address the issue of WSS estimation in its entirety. Indeed, a WSS estimation technique must be able to work in the following two situations:

- (S_{more}) the VM is wasting memory,
- (S_{less}) the VM is lacking memory.

The *VMware* technique [12] is only appropriate in (S_{more}) while *Geiger* and *Hypervisor exclusive cache* are effective in (S_{less}). Only *Zballoond* and *self-ballooning* cover both (S_{more}) and (S_{less}). Our study also shows that each solution comes with its strengths and weaknesses. The next section presents our solution.

6 Badis

6.1 Presentation

The previous section shows that the WSS estimation problem is addressed by a wide range of solutions. However, to the best of our knowledge, none of them are consistently adopted in the mainstream cloud. We assert that one reason which leads the cloud customers to the denial of such solutions is their intrusiveness (both from the codebase and from the performance perspective). This is confirmed by our cloud partner, Eolas [49]. We claim that a solution easily adopted in the mainstream cloud should provide (1) no codebase intrusiveness and (2) low performance impact. In order to reduce the performance impact the solution should provide high accuracy and thereby, address both (S_{more}) and (S_{less}).

This section presents *Badis*, a system which smartly combines existing techniques in such a way that both (S_{more}) and (S_{less}) are covered with no codebase intrusiveness. Indeed, we found that even if the *VMware* and *Geiger* solutions have a fairly high performance impact they have no intrusiveness in the VM's codebase. The second observation is that these solutions are complementary (*VMware* addresses S_{more} while *Geiger* addresses S_{less}). The *Hypervisor exclusive cache* is also a solution that only addresses (S_{less}) but it has higher *hyper_over*. Thereby, a system which is able to combine *VMware* and *Geiger* satisfies our all requirements.

Fig. 7 top presents the architecture of our system. The *VMware* technique is implemented at the hypervisor level while *Geiger* as well as the feedback loop decision module are located inside the dom0. Concerning the *VMware* technique, we rely on the accessed bit instead of the present bit for memory page invalidation. The former introduces less overhead on the VM than the latter. The decision module is implemented as a kernel module inside the dom0, thus keeping the hypervisor as lightweight as possible. The communication between

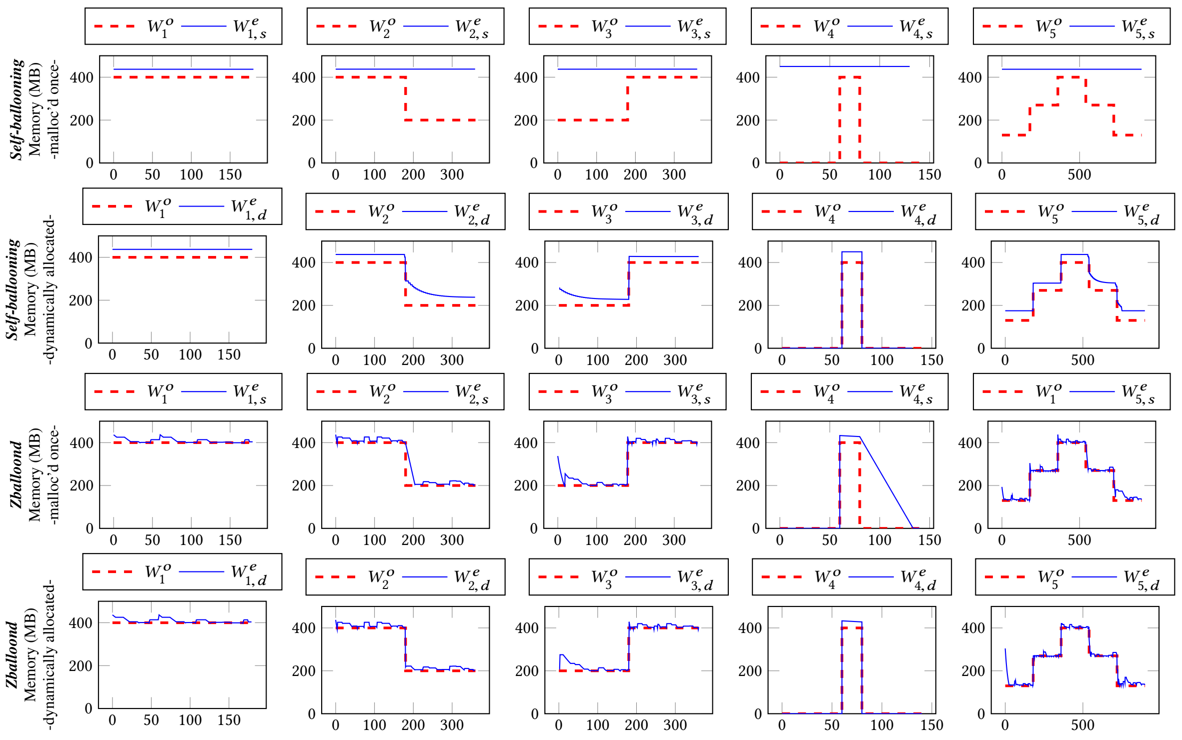

Figure 5. Evaluation results of *self-ballooning* and *Zballoond* with synthetic workloads. The original workload is noted W_i^o while the actual estimated WSSs are noted W_{ij}^e. "j" is s (the static implementation) or d (the dynamic implementation).

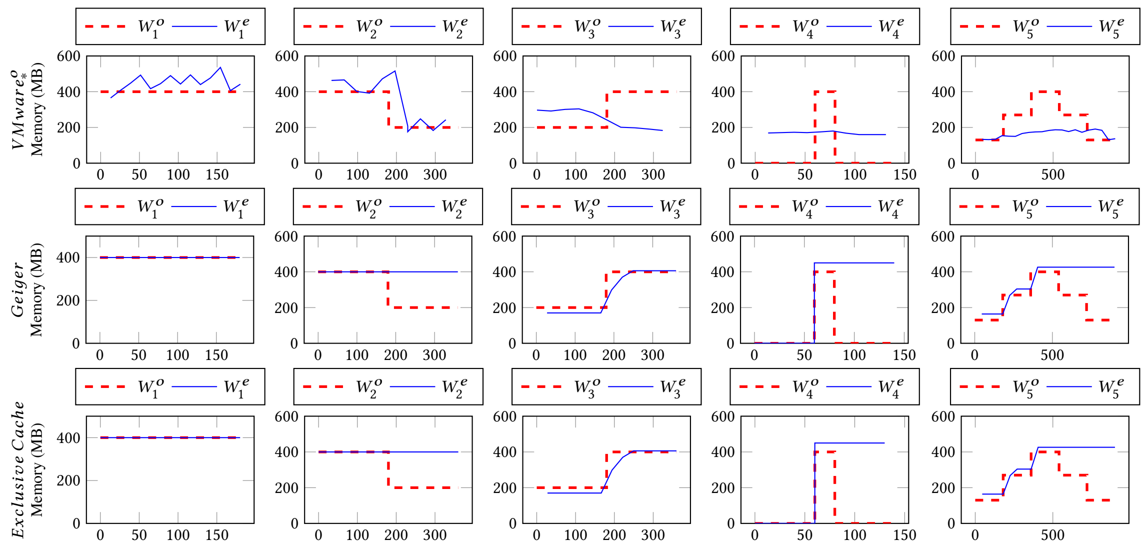

Figure 6. Evaluation results of *VMware*[2], *Geiger*, and *Exclusive cache* with synthetic workloads.

Geiger and the decision module is straightforward since they both run inside the dom0. Concerning the *VMware* technique, it communicates with the decision module via a shared memory established between the dom0 and the hypervisor. To this end, we extend the native Xen `share_info` data structure, which implements the shared memory used by the hypervisor to provide the VM with hardware information necessary at VM boot time (e.g. the memory size). Having described the mechanisms which allow the global functioning of our system, let us now present how the two WSS estimation techniques are leveraged.

[2]The accuracy of the *VMware* method is orthogonal to the implementation approach thereby, it is represented only once.

		Self-ballooning		*Zballoond*		$VMware_{present}$	
Benchmark and app.		**vm_over**	**hyp_over**	**vm_over**	**hyp_over**	**vm_over**	**hyp_over**
Dacapo	avrora	1	1	1.19	1	2.77	1.06
	batik	1	1	1.09	1	15.44	2.0
	eclipse	1	1	3.67	1	18.79	1.01
	h2	1	1	2	1	24.12	2.05
	jython	1	1	1.58	1	21.42	1.16
Cloud suite	Data Anal.	1	1	1.4	1	45.05	2.06
LinkBench	MySQL	1	1	2.92	1	20.17	1

		$VMware_{access}$		*Geiger*		*Exclusive Cache*	
Benchmark and app.		**vm_over**	**hyp_over**	**vm_over**	**hyp_over**	**vm_over**	**hyp_over**
Dacapo	avrora	2.14	1.1	1.22	1.2	1	5.06
	fop	13.06	2.2	1.41	1.32	1.5	5.6
	h2	15.63	1	1	1.02	1	5.0
	jython	20.51	2	1.12	1.5	1.7	4.9
	luindex	18.2	1.5	1.04	1.45	1.08	5.52
Cloud suite	Data Anal.	40.22	1.06	1.15	1.22	2.03	6.04
LinkBench	MySQL	19.22	2	1.76	1.09	1.80	5.2

Table 1. Evaluation results of each technique with macro-benchmarks.

	Self-b.	*Zballoond*	*VMware*	*Geiger*	*Excl. Cache*
intrusive	yes	yes	no	no	no
active	no	yes	yes	no	yes
addressed situations	all	all	S_{more}	S_{less}	S_{less}

	Self-b.	*Zballoond*	*VMware*	*Geiger*	*Excl. Cache*
accuracy	depends on the app.	high	high in S_{more} zero in S_{less}	high in S_{less} zero in S_{more}	high in S_{less} zero in S_{more}
vm_over	nil	almost nil	nil in S_{more} high in S_{less}	almost nil	almost nil
hyper_over	nil	nil	almost nil	almost nil	not negligible

Table 2. Study synthesis of all WSS estimation techniques according to both qualitative (left) and quantitative (right) metrics.

For each VM, the system implements a 3-state finite state machine (FSM), as shown in Fig. 7 bottom. Once setup, the VM enters the V state in which the WSS is estimated using the *VMware* technique (*Geiger* is disabled). In fact, it is more likely that the memory allocated to the VM at boot time (booked by its owner) is larger than its WSS. While in the V state, if the estimated WSS moves closer[3] to the VM's allocated memory, the FSM transitions to the VG state in which *Geiger* is enabled. While in the VG state, the WSS of the VM is given by the *VMware* technique if *Geiger* does not measure any swap activity. Otherwise, the WSS is given by *Geiger*. The FSM transitions from VG to the G state (in which the *VMware* technique is disabled) when *Geiger* reports swap activities during several consecutive rounds. Finally, the transition from G to V is triggered if *Geiger* does not observe any swap activity during several consecutive rounds. We have observed that two consecutive rounds are enough to filter the memory spikes and steep valleys.

One may doubt the need of VG state. However, we consider it necessary because of a more subtle VMware limitation. As presented before, VMware chooses a set of sample pages and based on the number of pages accessed during an observation interval, it computes the WSS as a percentage of the total memory. For example, if VMware chooses 100 sample pages and 60 of them are accessed, it concludes that the WSS size is 60% of the total VM's memory. However, in most of the cases this is wrong and not only because of the estimation error. The VMware technique considers all pages equal and swappable. Nevertheless, some of the pages are pinned down by the OS. If they are not accessed during a VMware observation interval, they are considered out of the working set. When the memory is adjusted to the WSS the OS cannot swap out these pinned pages and thereby, it has to chose from the active pages. This issue is an important source of performance degradation.

Further we will present how Badis cope with this problem. When in VG, the VM is in a swapping state which means that all of its allocated memory is necessary. In this state we still continue to read estimations from the *VMware* technique which theoretically should be 100% (i.e. all pages are accessed during a time frame). However, the estimations are generally less than 100% (e.g. 80%) because of the pinned pages which are inactive. The difference to 100% (e.g. 20%) should also be included in the working set because, even if these pages are inactive, they cannot be swapped-out. This correctional value is stored and leveraged later, in the V state, for a conclusive estimation.

The choice of the estimation time slice depends on each technique. The *VMware* technique is the most sensitive one because the percentage of pages "touched" during the estimation interval should be representative for the global memory activity. Badis employs the same time slice as the one proposed by [12] (i.e. 1 min). We have observed that a shorter time slice significantly increases the estimation error. On the other hand, *Geiger* is more resilient to shorter time slices. Thereby, Badis chooses a time slice of 30 sec for *Geiger*. The next section presents the way our estimation system is leveraged in a virtualized cloud.

6.2 Badis in a virtualized cloud

In the last section we presented the advantages of Badis over the state-of-the-art. However, one may ask which are the benefits of WSS estimation in the cloud? Clearly, there is no benefit in shrinking a VM's memory unless there is some other VM ready to make use of that. Thereby, the WSS estimation should be integrated in a higher level system that has a wide image on the datacenter's compute resources. Such a system is the cloud

[3] By default, the threshold is 80%.

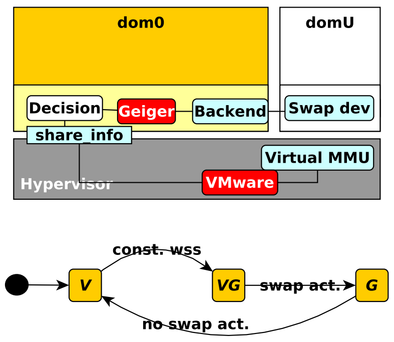

Figure 7. (top) The architecture of Badis. (bottom) The finite-state machine used to track a VM's WSS in Badis.

manager (e.g. OpenStack [38]). The latter controls the VMs' lifetime and takes consolidation decisions.

Generally, the factor that limits the server consolidation is memory, for two main reasons. First, over the last several years, we have seen new applications with vastly growing memory demands, while platform evolution continued to offer more CPU capacity growth than memory. This referred to as *the memory capacity wall* [8]. Second, in most of the virtualization systems, the booked memory (m_b) is entirely allocated when the VM is booted. This quantity should meet the highest possible memory demands the VM will have during its lifetime. However, most of the time, the memory demands are lower than m_b which implies some degree of memory waste (see Fig. 8). **The WSS estimation could help improving the memory efficiency and thereby, increase the consolidation ratio.** However, in some circumstances, the server consolidation based on the VMs' current WSS estimation may do more harm than good. If a recently consolidated VM requests more memory than available on the hosting server, it should be migrated back on a server which can provide enough memory. This excessive VM dynamics may increase the datacenter's energy consumption [35] and impact the hosted applications' performance [36]. Thereby, the research question is: how to leverage the WSS estimation techniques not only for a better but also for a stable consolidation? Further we will present our solution to this problem.

Our solution is implemented as an extension to a popular consolidation system, namely OpenStack Neat [37]. The latter takes consolidation decisions when a server is (1) underloaded or (2) overloaded. In the first case it relocates all VMs in order to free up the server and switch it to a lower energy state. In the latter case it migrates one VM, generally the one with the smallest allocated memory, to reduce the migration time. We mention that Neat places VMs based on the booked memory and not the WSS estimation. In order to decide when a server is underloaded or overloaded, Neat has a data collection module that fetches the CPU utilization of all VMs and stores the data in both, the local datastores on each physical server and a global datastore for the entire datacenter. However, since Neat does not overcommit memory, it does not collect any memory utilization data. The underload and overload detection algorithms only take into account the CPU. Further we will present how Badis adjusts a VM's allocated memory based on its WSS.

		Self-ballooning	*Zballoond*	Badis	
Benchmark and app.		**vm_over**	**vm_over**	**vm_over**	**hyper_over**
Dacapo	avrora	1	1.19	1.26	1.8
	batik	1	1.09	1.57	1.05
	eclipse	1	3.67	1	1.68
	h2	1	2	1.16	1.3
	jython	1	1.58	1.05	1.15
Cloud suite	Data Analytics	1.29	1.4	1.16	1.2
LinkBench	MySQL	1.11	2.92	1.09	1

Table 3. Evaluation of our solution with macro-benchmarks, and comparison with two existing solutions.

First, Badis continuously computes the moving average of the last n WSS estimation samples (e.g. $n = 5$). We monitor the moving average of each WSS using time slices of size s (e.g. $s = 1$ *hour*). The allocated memory of VM id_vm is adjusted to *the maximum value of the moving average in the last time slice*, noted $WSS_{id\_vm}^{max\_avg}$. The latter value is also transmitted to the data collection module (see Fig. 9). We have modified the Neat's underload and overload detection algorithms to also take into account the memory load and pack the VMs based on $WSS_{id\_vm}^{max\_avg}$. Since $WSS_{id\_vm}^{max\_avg} \leq m_b$, the VM packing is tighter. If the allocated resources of all VMs on a server overpasses the underload or the overload threshold, Neat will trigger a new consolidation round (see Fig. 11). However, the volatility of the memory load is generally lower than the CPU. In our experiments only 3% of the consolidation rounds were triggered because of the memory load (see Section 6.3).

6.3 Evaluations

The experimental environment is the same one presented in Section 5. We evaluated our solution with both micro and macro benchmarks.

Micro-benchmark based evaluations. We first validated the effectiveness of our solution using a synthetic workload, see the dashed blue curve in Fig. 10. This workload includes situations a WSS estimation technique should cope with. One can observe that the accuracy of our solution is comparable with Zballoond but without any VM codebase intrusiveness. In the last part of Fig. 10 we can observe a case where our solution even outperforms Zballoond: the WSS drops quickly and the inactive pages are still allocated. In this case Badis is able to quickly track the new WSS while Zballoond slowly decreases the WSS leading to a lot of resource waste.

Macro-benchmark based evaluations. We also evaluated our solution with macro-benchmarks, see Table 3. The latter focuses on the *hyper_over* and the *vm_over* metrics since the *accuracy* metric has been evaluated above. We compare our solution with the only solutions which address the issue of WSS estimation in its entirety, namely *self-ballooning* and *Zballoond*. We can see that our solution leads to a negligible overhead on both the VM and the hypervisor/dom0 (less than 2x).

Simulations on traces from a Google datacenter. In the last sections we have demonstrated the capability of our solution to follow the WS variation with high precision. This section will

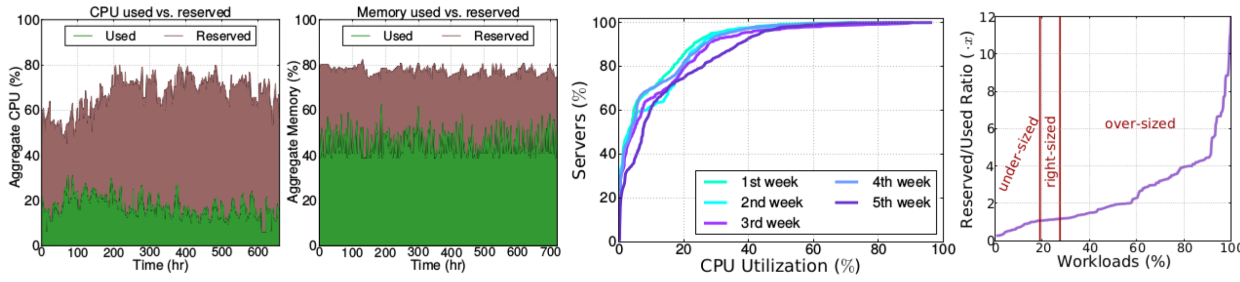

Figure 8. "Resource utilization over 30 days for a large production cluster at Twitter managed with Mesos. (a) and (b): utilization vs reservation for the aggregate CPU and memory capacity of the cluster; (c) CDF of CPU utilization for individual servers for each week in the 30 day period; (d) ratio of reserved vs used CPU resources for each of the thousands of workloads that ran on the cluster during this period." [6]

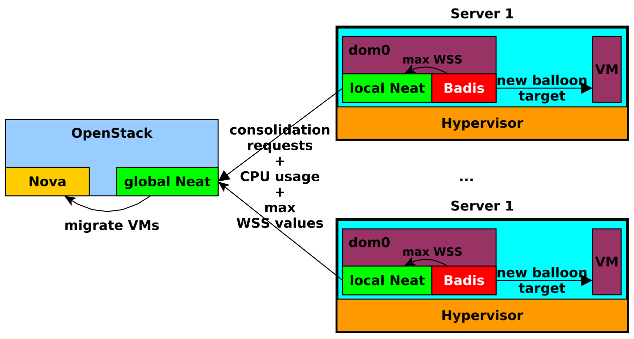

Figure 9. The integration of Badis in OpenStack. Badis estimates the WSS and sets the *id_vm*'s allocated memory to $WSS_{id\_vm}^{max\_avg}$. It also transmits $WSS_{id\_vm}^{max\_avg}$ values to the local Neat. The latter collects these values along with the CPU loads and sends them in batches to the global Neat. The local Neat may also send consolidation requests to the global Neat in the case of CPU/RAM overload/underload. These consolidation requests are decomposed into individual VM migrations which are executed by OpenStack Nova.

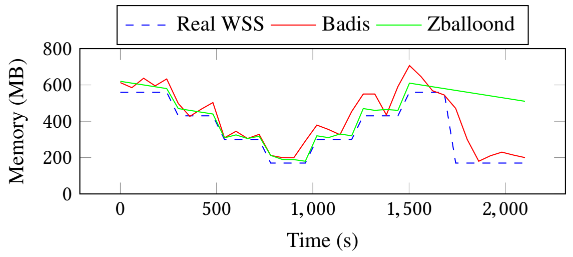

Figure 10. Badis and Zballoond evaluated with a synthetic workload.

show the effect of WSS estimation on the VM consolidation. In this respect, we leverage traces from a Google datacenter [40]. They represent the execution of thousands of jobs on a cluster of about 12,5k servers, monitored for about 29 days. Each job can be composed of several tasks and each task runs inside a container. For each container, the traces provide data such as the creation time, the destruction time, the amount of CPU/memory requested at creation time. Moreover the traces provide the amount of CPU/memory actually assigned to the container[4]. By relying on GloudSim [39] (a cloud simulator with VMs based on Google traces) we have simulated both, a consolidation based on the booked memory and a consolidation based on the actually assigned memory. In the first case the datacenter has an average of 9562 active servers while in the second case the average number of active servers is 4676. These figures prove that the memory is indeed the resource which limits the VM consolidation. In the second consolidation type, the packing ratio is more than 2x higher. Regarding the VM dynamics, there were executed around 2.5M migrations in total. Only 75k migrations (i.e. 3.17%) were caused by memory overload/underload. These results prove that the memory volatility is net inferior to the CPU volatility. However, the paradox is that most of the popular consolidation systems overcommit CPU but not RAM memory. Our evaluation results are totally reproducible using the code provided at [1].

7 Related work

The reader should refer to Section 4 for the presentation of the main WSS estimation techniques in virtualized environments. In this section we focus on other studies related to the concept of WSS, memory management and VM consolidation in a virtualized datacenter.

Working set size estimation. WSS estimation [24] could require large data collection and complex processing. Weiming Zhao et al. [23] have introduced a working set size estimation system which computes a VM's WSS based on its miss-ratio curve (MRC). The latter shows the fraction of the cache misses that would turn into cache hits if the VM's allocated memory increases. Moreover, Weiming Zhao et al. have evaluated the overhead of their solution by providing the relationship between performance and allocated memory size. Pin Zhou et al. [33] have proposed two similar methods which dynamically track the MRC of applications at run time. These techniques represent the hardware and the software implementations of the

[4]The sampling time interval for this data is around 5 minutes.

Mattsons stack algorithm. The latter relies on a ”stack” which stores the references to accessed pages (the most recently used page is on the top of the stack). Similarly to the ghost buffer, this algorithm computes the miss ratio curve based on the distance to the top of the stack. Carl Waldspurger et al [34] have proposed an approximation algorithm that reduces the space and time complexity of reuse-distance analysis. This algorithm is appropriate for online MRC generation due to its modest resource requirements.

Memory optimization techniques. Memory deduplication is one of the most popular memory optimization techniques. It consists in merging identical memory pages by keeping only one copy of it. This is mostly useful in case of read-only pages that stay unchanged during the VM run time. Depending on the algorithm used to identify similar pages, there are several implementations of page sharing [9, 12, 25, 26]. These techniques are often combined with memory compression tools to achieve better optimization rates [30–32]. Another memory optimization tool is the transcendent memory [29] which gathers the VMs' idle memory and the VMM non-allocated memory to a common pool.

Memory balancing is a memory optimization technique, that tries to adjust the VM's allocated memory depending on its necessities. Memory ballooning is the main concept behind this approach. The balancing techniques typically rely on working set size estimation techniques to optimize the memory usage [22]. In a latter work, Zhao et al. [28] leverages inexpensive working set tracking systems to correctly estimate the working set size for the Memory Balancer (MEB) [22]. Xiaoqiao Meng et al. [41] leverage the concept of statistical resource multiplexing between multiple VMs. Specifically, this paper proposes to form pairs of VMs that have complementary temporal behavior (i.e. the peaks of one VM coincide with the valleys of the other). Thereby, if consolidated together, the unused resources from the VM with low demands could be lent to the VM with high demands. These pairs of VMs are found out by computing the correlation between all combinations of two VMs in the datacenter. As one can notice, this approach requires high amount of computation even for small datacenters.

Improving Memory balancing drawbacks. Memory balancing techniques have several drawbacks. First, in the case where several VMs reach their respective memory limit simultaneously, they will all generate a high amount of I/O requests which may saturate the secondary storage. On the other hand, memory balancing is not aware of the hosted applications. Thus, memory intensive applications (e.g. database engines) face serious issues because of memory balancing techniques. To overcome these issues, [27] extends the VM memory ballooning to user level, for applications that manage their own memory.

VM consolidation. The VM consolidation is an NP hard problem [42]. Thereby, numerous papers came up with heuristics for this problem [43–46]. However, few of these projects provide real implementations to the proposed algorithms [37, 47]. Among the implemented systems, to the best of our knowledge, no system consistently performs memory overcommitment. Even if memory is the main consolidation impediment, most of the existing systems consolidate the VMs based on their booked memory and not on the actually used memory. In this paper, we propose a system that monitors the WSS of VMs and takes consolidation decisions based on the observed memory utilization.

8 Conclusion

In this paper, we present a systematic review of the main WSS estimation techniques, namely *Self-ballooning*, *Zballoond*, *VMware*, *Geiger* and *Hypervisor exclusive cache*. From far of our knowledge, this is the first work which deeply compares existing WSS techniques. To this end, we have proposed a set of qualitative and quantitative metrics allowing the classification of these techniques. We have evaluated each technique using both micro and macro benchmarks. The evaluation results revealed the strengths and the weaknesses of each technique. More important, they show that not all solutions address the issue in its entirety. Unfortunately, those which entirely address the issue are intrusive, thus requiring the permission of the VM's owner. This is unacceptable from the datacenter operator's point of view. We also propose Badis, a system which combines several of the existing solutions, using the right solution at the right time. In addition, we have implemented a consolidation extension which leverages Badis for an improved consolidation ratio. The simulations performed on traces from a Google datacenter reveal a 2x better consolidation ratio with only 3% additional VM migrations.

9 Acknowledgements

We would like to thank Vishal Misra (our shepherd) and the anonymous reviewers for their helpful feedback. This work benefited from the support of Europe through the Erasmus+ International Credit Mobility program.

Appendices

A The Global Neat

Figure 11 presents the consolidation algorithm employed by global Neat (i.e. Best Fit Decreasing). The VMs are sorted in descending order based on their CPU load and WSS (line 3) while the hosts (physical machines) are sorted in ascending order based on their available resources (line 4). In every datacenter, a part of the hosts are inactive (i.e. suspended-to-RAM or shut-down); they are also sorted in ascending order[5] (line 5). Subsequently, Neat tries to find, for each VM, a host that have enough available resources (lines 7-16). If no host is able to accommodate the VM, Neat wakes up an inactive host (lines 17-25). Finally, if all VMs are assigned to corresponding hosts, Neat transmits the mapping list to Nova which actually executes the VM migrations (lines 26-27).

[5]The datacenter may be heterogeneous (i.e. different types of hosts).

```
#host_tuple=(host_cpu, host_mem, host_id); vm_tuple=(vm_cpu, $WSS_{vm\_id}^{max\_avg}$, vm_id)
def global_neat(hosts_tuple, inactive_hosts_tuple, vms_tuple):
        vms = sort_decreasing(vms_tuple)
        hosts = sort_increasing(hosts_tuple)
        inactive_hosts = sort_increasing(inactive_hosts_tuple)
        mapping = {}
        for vm_cpu, vm_ram, vm_id in vms:
                mapped = False
                while not mapped:
                        for host_id in hosts:
                                if hosts_cpu[host_id] >= vm_cpu and hosts_ram[host_id] >= $WSS_{vm\_id}^{max\_avg}$:
                                        mapping[vm_id] = host
                                        hosts_cpu[host_id]  = vm_cpu
                                        hosts_ram[host_id]  = vm_ram
                                        mapped = True
                                        break
                        if not mapped:
                                if inactive_hosts:
                                        activated_host = inactive_hosts.pop(0)
                                        hosts.append(activated_host)
                                        hosts = sort_increasing(hosts)
                                        hosts_cpu[activated_host[2]] = activated_host[0]
                                        hosts_ram[activated_host[2]] = activated_host[1]
                                else:
                                        break
        if len(vms) == len(mapping):
                return mapping
        return {}
```

Figure 11. The global Neat consolidation algorithm

References

[1] https://github.com/papers02/working_set.git

[2] U. Hölzle and L. André Barroso. The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines. *Morgan and Claypool Publishers*, 2009.

[3] Americas Data Centers Are Wasting Huge Amounts of Energy. http://anthesisgroup.com/wp-content/uploads/2014/08/Data-Center-IB-final826.pdf

[4] C. Subramanian, A. Vasan, and A. Sivasubramaniam. Reducing data center power with server consolidation: Approximation and evaluation. *HiPC*, 2010.

[5] L. Andr Barroso and U. Hlzle The Case for Energy-Proportional Computing. IEEE Computer 2007.

[6] C. Delimitrou and C. Kozyrakis Quasar: resource-efficient and QoS-aware cluster management. ASPLOS 2014.

[7] D. Meisner, B. T Gold, and T. F Wenisch The PowerNap Server Architecture. ACM Transaction on Computer Systems 2011.

[8] K. T. Lim, J. Chang, T. N. Mudge, P. Ranganathan, S. K. Reinhardt, T. F. Wenisch Disaggregated memory for expansion and sharing in blade servers. ISCA 2009.

[9] G. Mis, D. G. Murray, S. Hand, and M. A. Fetterman Satori: enlightened page sharing. ATC 2009.

[10] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, A. Warfield Xen and the Art of Virtualization. SOSP 2003.

[11] Amazon Web Services, Inc. https://aws.amazon.com/ec2/

[12] C. A. Waldspurger Memory Resource Management in VMware ESX Server. OSDI 2002.

[13] https://blog.xenproject.org/2008/08/27/xen-33-feature-memory-overcommit/. visited on May 2017.

[14] https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Deployment_Guide/s2-proc-meminfo.html. visited on May 2017.

[15] J. Chiang, L. Han-Lin, and C. Tzi-cker. Memory Working Set-based Physical Memory Ballooning. ICAC 2013.

[16] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Geiger: monitoring the buffer cache in a virtual machine environment. SIGARCH 2006.

[17] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. SOSP 1995.

[18] P. Lu and K. She. Virtual machine memory access tracing with hypervisor exclusive cache. ATC 2007.

[19] Blackburn, S. M., Garner, R., Hoffman, C., Khan, A. M., McKinley, K. S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S. Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J. E. B., Phansalkar, A., Stefanovic, D., VanDrunen, T., von Dincklage, D., and Wiedermann, B. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications, (Portland, OR, USA, October 22-26, 2006)

[20] CloudSuite. http://cloudsuite.ch/. visited on May 2017.

[21] T. G. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan LinkBench: a database benchmark based on the Facebook social graph. SIGMOD 2013.

[22] W. Zhao and Z. Wang Dynamic memory balancing for virtual machines. VEE 2009.

[23] W. Zhao, X. Jin, Z. Wang, X. Wang, Y. Luo, and X. Li Low cost working set size tracking. ATC 2011.

[24] Melekhova A, Markeeva L. Estimating Working Set Size by Guest OS Performance Counters Means. The Sixth International Conference on Cloud Computing, GRIDs, and Virtualization 2015.

[25] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum Disco: Running Commodity Operating Systems on Scalable Multiprocessors. ACM Trans. Computer Systems, vol. 15, no. 4, pp. 412-447, 1997.

[26] Gupta D, Lee S, Vrable M, Savage S, Snoeren C A , Varghese G, Voelker M. G, Vahdat A Difference Engine: Harnessing Memory Redundancy in Virtual Machines OSDI 2008.

[27] Tudor-Ioan Salomie, Gustavo Alonso, Timothy Roscoe, and Kevin Elphinstone. Application level ballooning for efficient server consolidation. EuroSys 2013.

[28] Weiming Zhao Zhenlin Wang. Dynamic Memory Balancing for Virtualization. TACO 2016.

[29] Dan Magenheimer, Chris Mason, Dave McCracken, Kurt Hackel. Transcendent Memory and Linux. Ottawa Linux Symposium (OLS) 2009

[30] Irina Chihaia Tuduce and Thomas Gross. Adaptive Main Memory Compression. ATC 2005.

[31] Gennady Pekhimenko, Todd C. Mowry, and Onur Mutlu. Linearly Compressed Pages: A Main Memory Compression Framework with Low Complexity and Low Latency. PACT 2012.

[32] Lei Yang Haris Lekatsas Robert P. Dick. High-Performance Operating System Controlled Memory Compression. DAC 2006.

[33] Pin Zhou, Vivek Pandey, Jagadeesan Sundaresan, Anand Raghuraman, Yuanyuan Zhou and Sanjeev Kumar. Dynamic tracking of page miss ratio curve for memory management. ASPLOS 2004.

[34] Carl A. Waldspurger, Nohhyun Park, Alexander Garthwaite, and Irfan Ahmad. Efficient MRC Construction with SHARDS. FAST 2015.

[35] Haikun Liu, Cheng-Zhong Xu, Hai Jin, Jiayu Gong, Xiaofei Liao Energy modeling for live migration of virtual machines. Cluster Computingi.

[36] William Voorsluys, James Broberg, Srikumar Venugopal, Rajkumar Buyya Cost of Virtual Machine Live Migration in Clouds: A Performance Evaluation. CloudCom.

[37] Anton Beloglazov, Rajkumar Buyya OpenStack Neat: a framework for dynamic and energy-efficient consolidation of virtual machines in OpenStack clouds. Concurrency and Computation: Practice and Experience.

[38] Omar Sefraoui, Mohammed Aissaoui, Mohsine Eleuldj Openstack: Toward an open-source solution for cloud computing. International Journal of Computer Applications.

[39] Sheng Di, Franck Cappello. GloudSim: Google trace based cloud simulator with virtual machines. SPE 2015.

[40] Google Traces. https://github.com/google/cluster-data/blob/master/ClusterData2011_2.md

[41] Meng, Xiaoqiao and Isci, Canturk and Kephart, Jeffrey and Zhang, Li and Bouillet, Eric and Pendarakis, Dimitrios. Efficient Resource Provisioning in Compute Clouds via VM Multiplexing. ICAC 2010.

[42] A. Karve, T. Kimbrel, G. Pacifici, M. Spreitzer, M. Sviridenko, and A. Tantawi. Dynamic placement for clustered web applications. In *WWW*, 2006.

[43] K.H.Kim, A.Beloglazov, R.Buyya. Power-aware provisioning of cloud resources for real-time services. MGC 2009.

[44] A.Beloglazov, R.Buyya. Energy efficient resource management in virtualized cloud datacenters. Cloud and Grid Computing 2010.

[45] H.S. Abdelsalam, K. Maly, R. Mukkamala, M. Zubair, D. Kaminsky. Analysis of energy efficiency in Clouds. Computation World: Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns, 2009.

[46] G.Jung, M.A.Hiltunen, K.R.Joshi, R.D.Schlichting, C.Pu Mistral:dynamically managing power, performance, and adaptation cost in Cloud infrastructures. ICDCS 2010.

[47] Feller E, Rilling L, Morin C. Snooze: a scalable and autonomic virtual machine management framework for private clouds. CCGrid 2012.

[48] Jinchun Kim, Viacheslav Fedorov, Paul V. Gratz, A. L. Narasimha Reddy Dynamic Memory Pressure Aware Ballooning. MEMSYS 2015.

[49] Eolas cloud provider. https://www.eolas.fr/