SmarTmem: Intelligent Management of Transcendent Memory in a Virtualized Server

Luis A. Garrido
Barcelona Supercomputing Center
luis.garrido@bsc.es

Rajiv Nishtala
Norwegian University of Science and Technology
rajiv.nishtala@ntnu.no

Paul Carpenter
Barcelona Supercomputing Center
paul.carpenter@bsc.es

***Abstract*—Managing memory capacity in virtualized environments is still a challenging problem. Many solutions have been proposed and implemented, including memory ballooning and memory hotplug. But these mechanisms are slow to respond to changes in virtual machine (VM) memory demands. Transcendent Memory (tmem) was introduced to improve responsiveness in memory provisioning, by pooling idle and fallow memory in the hypervisor, and making these physical pages available as additional memory for the VMs through a key–value store.**

However, tmem presents some limitations of its own. State-of-the-art hypervisors do not implement any efficient way to manage tmem capacity, letting VMs compete for it in a greedy way by default, regardless of their actual memory demand.

In this paper, we demonstrate the need for intelligent memory capacity management for tmem, and we present the design and implementation of SmarTmem, a mechanism that integrates coarse-grained user-space memory management with fine-grain allocation and enforcement at the virtualization layer. Our results show that our solution can improve the running time of applications from the Cloudsuite benchmarks by up to 35% compared to the default tmem allocation mechanism.

I. Introduction

Virtualization technology is prevalent across cloud service providers as it reduces capital and operational costs [1]–[6]. Each physical server, or computing node, executes a Virtual Machine Monitor (VMM) or hypervisor [7], [8], which creates virtual machines (VMs) to run the guest OSes, and manages the physical resources allocated to the VMs. The hypervisor multiplexes the CPUs and I/O devices and it controls the allocation of the physical memory capacity. In such environments, memory is often one of the most critical and scarce resources [9], [10].

State-of-the-art hypervisors have multiple mechanisms to dynamically manage memory capacity [11]–[13]. Among these is Transcendent Memory [13], which was introduced to improve memory reallocation responsiveness, since other mechanisms like memory ballooning are relatively slow to reallocate memory upon changes in memory demand [13]. Tmem works by centralizing memory management within the hypervisor through a key–value store. Tmem has two modes of operation: 1) *frontswap*, which serves as a page cache for pages swapped out by the VMs, and 2) *cleancache*, which serves as a page cache for clean pages that were fetched from disk. In either mode of operation, tmem pools all the idle and fallow (unassigned to VMs) memory pages in the node, and the pages get assigned to the VMs as tmem when they need it. If there are no free pages, any write to a tmem page will fail, causing an access to the (virtual) disk device.

When multiple VMs use tmem, the hypervisor will assign pages to the VMs in a greedy manner. The VMs compete for the tmem capacity by default, so if some VMs take a large amount of the available tmem, the rest of VMs will starve and will generate a large number of disk accesses, degrading performance across the VMs and the computing node [14].

In this paper, we address the issue of optimizing VM memory allocation using Transcendent Memory in a single computing node. We introduce SmarTmem, a software stack for intelligent tmem allocation in a single node, created by re-purposing the software-stack developed in [15]. Instead of using tmem for pooling memory capacity across multiple nodes (as in [15]), this paper analyzes in depth the problem of tmem optimization at the single node level. To the best of our knowledge, this is the first effort to analyze tmem capacity optimization in a single virtualized computing node using a software-stack with the architecture of SmarTmem.

SmarTmem employs a user-space process executing in Xen's privileged domain that implements intelligent tmem management policies based on the memory utilization behavior of the VMs. Our results show that using high-level policies improves the running time of applications compared to the baseline implementation of tmem.

To summarize, the main contributions of this paper are:
(1) We demonstrate that the default way of allocating tmem at the single node level, used in state-of-the-art hypervisors, is unable to adapt to changes in memory demand and to ensure fair and proportional allocation of tmem when multiple VMs are active.
(2) We re-purpose the architecture in [15] to create SmarTmem, a software architecture for intelligent memory management for tmem in a single virtualized computing node.
(3) We implement high-level tmem management policies in SmarTmem and evaluate them using benchmarks from CloudSuite [16], and our results show up to 35% improvement over the default greedy approach of allocating tmem.

The rest of this paper is organized as follows. Section 2 provides background on virtualization technologies, tmem and its implementation in Xen. Section 3 discusses the SmarTmem architecture and the high-level tmem management policies. Section 4 describes our experimental framework. Section 5 presents the evaluation results. Section 6 expands on the related work. Finally, Section 7 states our conclusions and future work.

II. Background

This section provides a background on virtualization technology and the state-of-the-art memory management mechanisms. Additionally, it provides insight into Tmem, its functionality and drawbacks.

A. Virtualization and Memory Management

Cloud services are built on top of virtualization technology that rely on a hypervisor to multiplex physical resources such as CPUs and I/O interfaces and allocate others, such as memory. When a new VM is created, it is allocated a portion of the physical memory. If the VM increases its demand for memory capacity, i.e. its memory capacity becomes under-provisioned, it will start accessing its (virtual) disk device(s), even if some physical memory may be available in the computing node, either unallocated by the hypervisor *(fallow)* or allocated to another VM that does not need it *(idle)*. When a VM has idle memory, the VM is overprovisioned of memory capacity. In general, it is necessary to continuously readjust the allocation of memory to VMs, in order to optimize memory utilization.

State-of-the-art hypervisors such as Xen, dynamically reallocate memory among the VMs using memory ballooning and memory hotplug. Both of these mechanisms are slow to respond to rapid changes in memory demand, requiring successful prediction of memory demand behavior or overprovisioning of physical memory in the hypervisor [13].

B. Transcendent Memory (tmem)

Transcendent memory (tmem) was introduced to overcome the limitations of memory ballooning and memory hotplug [13]. Tmem pools together all the idle/underutilized and fallow memory in the node, and allocates it to the VMs through a page-copy–based interface providing a key–value store with synchronous *put*, *get* and *flush* operations. The capacity of transcendent memory and the physical locations of data stored in it are unknown to the guest VMs.

Xen's tmem can be used by Linux in two modes, *cleancache* and *frontswap*. Linux cleancache is a victim cache for clean pages that are evicted by the Linux kernel's Pageframe Replacement Algorithm (PFRA). Linux frontswap uses tmem as a cache in front of a swap device, so a successful store to tmem avoids a disk write and a later read.

In order to use tmem, each VM needs to have a kernel module with the tmem implementation in either cleancache or frontswap modes. Upon the module initialization, a pool of tmem memory pages is created for the VM, which remains under hypervisor control and is given an identifier. Every tmem page is identified by a three-element tuple (its key), consisting of the pool identifier, a 64-bit object identifier and a 32-bit offset or page identifier [13]. The object identifier and offset are both extracted by the kernel from the address of the page.

Figure 1 illustrates the operation of cleancache and frontswap, where VM-j and VM-i *put* and *get* a page, respectively. When the Linux kernel evicts a page from memory, due to high memory pressure, it will generate a page fault and the kernel will first attempt to write the page to tmem.

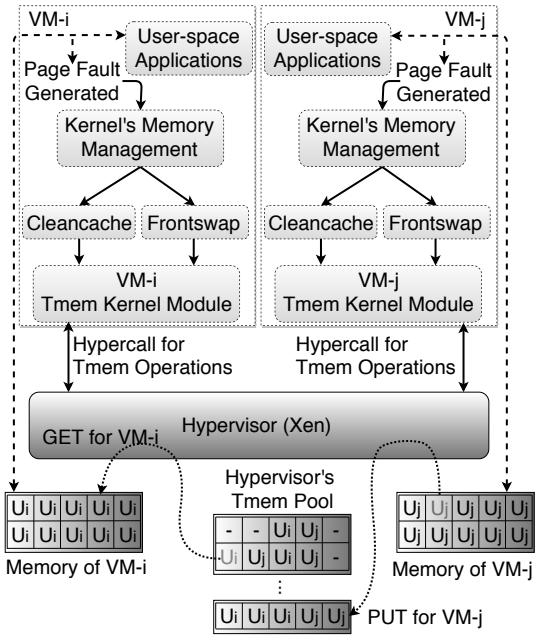

Fig. 1: Using tmem in Linux and Xen. Each VM accesses tmem pages via hypercalls.

To achieve this, the kernel traps the fault and passes it on to a tmem kernel module that initiates the tmem *put* hypercall. Once the hypecall reaches the hypervisor, it will allocate a physical page, if available, and will copy the data from the VM's memory into tmem. When the hypercall returns, the data has already been copied into tmem, so the guest OS can already reuse the memory for a different purpose.

When a user process attempts to access a page in tmem, the access will cause a page fault into the guest OS's kernel. The kernel will trap this fault once again, will send it to the kernel module that will generate a *get* hypercall for tmem. If the hypervisor can locate the page in tmem, it will copy its contents into the VM's pseudophysical address space.

The tmem backend also supports two flush operations: *flush page* and *flush object*. These operations are generated when a VM wishes to invalidate a tmem page or a group of them identified with the pool or object identifier, respectively. When this happens, the tmem page or pages are freed by the domain and can be used to *put* different VM page(s).

Current implementations of tmem allocate pages on *put*s in a greedy way, as long as there are free tmem pages. Thus, it is possible for one VM to acquire all the tmem pages if its maximum tmem allocation is not limited somehow. If one VM is able to take all the tmem pages, the rest of the VMs will incur in high-latency accesses to the I/O devices, significantly reducing their performance. As we shall see, it is necessary to intelligently manage the tmem capacity, in order to improve fairness, reduce the worst-case and overall running times.

III. SmarTmem: Optimizing Tmem Utilization

This section describes the architecture of SmarTmem, repurposed from [15], with the fundamental difference that

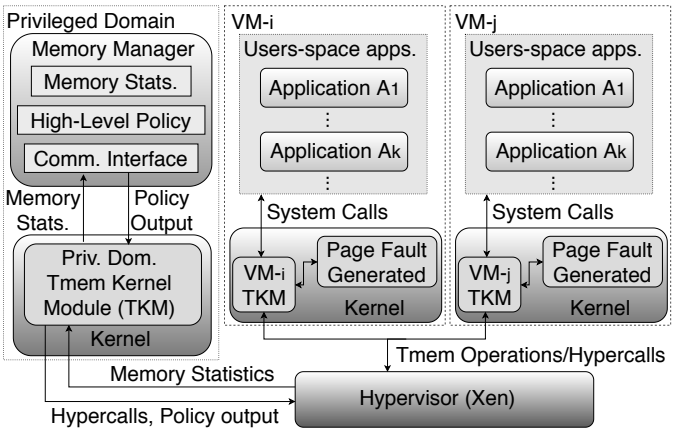

Fig. 2: SmarTmem architecture. The Memory Manager (MM) runs in Xen's privileged domain.

Algorithm 1 Tmem allocation in the Hypervisor

```
1:  function HYPERVISOR_OP(vm_data_hyp, id, op)
2:      tmem_used ← vm_data_hyp[id].tmem_used
3:      mm_target ← vm_data_hyp[id].mm_target
4:      if op == PUT then
5:          if tmem_used ≥ mm_target then
6:              return_value ← E_TMEM
7:          else if node_info.free_tmem == 0 then
8:              return_value ← E_TMEM
9:          else
10:             allocate_tmem_page(id)
11:             vm_data_hyp[id].tmem_used ← tmem_used + 1
12:             vm_data_hyp[id].puts_succ ← puts_succ + 1
13:             return_value ← S_TMEM
14:         end if
15:         vm_data_hyp[id].puts_total ← puts_total + 1
16:     else if op == FLUSH then
17:         deallocate_tmem_page(id)
18:         vm_data_hyp[id].tmem_used ← tmem_used − 1
19:         return_value ← S_TMEM
20:     end if
21:     return return_value
22: end function
```

SmarTmem focuses on the intelligent tmem management in a single virtualized computing node. SmarTmem divides tmem management between coarse-grain level memory management in a user-space process running in a privileged domain and fine-grain allocation and enforcement in the hypervisor.

A. SmarTmem Architecture

Figure 2 shows the architecture of SmarTmem, consisting of three components:

- Hypervisor support for SmarTmem
- Tmem Kernel Module (TKM)
- Memory Manager (MM) User-space Process for Tmem Allocation in a Single Virtualized Computing Node

B. Hypervisor support for SmarTmem

This section describes the hypervisor support for SmarTmem. The role of the hypervisor is to monitor the metrics used by the high-level policy, enforce target allocations and make the fine-grained tmem page allocation.

It is necessary for the hypervisor to gather information about the tmem capacity utilization by the VMs. This information is used by the MM to determine the tmem dynamic re-allocation. Table I summarizes the data collected by the hypervisor. SmarTmem requires to gather less information in the hypervisor than [15], since its scope is within a single computing node. It does not require to identify nodes or to keep track of the VMs in different nodes. Thus, the communication demands from the hypervisor to user-space are reduced for SmarTmem.

When a VM maxes out its memory, any future access that attempts a swap to disk will generate a *put* operation to tmem. The hypervisor monitors the amount of *puts* of each VM, as well as other tmem operations, similar to [15]. The hypervisor also keeps track whenever *puts* fail, and the amount of memory used by the VMs. When a *put* fails, it means that there is no tmem capacity to satisfy the request. Everytime this happens, the VM that generated the *put* will swap the page to disk.

The hypervisor gathers and monitors all the memory utilization behavior and sends it to the TKM in the privileged domain via a virtual interrupt request (VIRQ). This VIRQ is sent to the TKM every second, and the TKM passes it to the user-space MM process. With this information, the MM calculates new target tmem capacities for every VM and sends these targets back to the hypervisor again through the TKM, which generates a specific hypercall for this purpose.

Algorithm 1 presents pseudo-code to illustrate the tasks performed by the hypervisor. When the target allocations reach the hypervisor, it stores them and keeps them until the MM modifies them (line 3). Everytime a VM attempts to get a tmem page (through a *put*, line 4), the hypervisor checks if the current amount of tmem used by the VM is less than its target (line 5). If so, the VM obtains a page from the tmem pool, the hypervisor allocates a tmem page and copies the data contained in the VM's page into the tmem page frame (line 10). The allocation of pages in SmarTmem is different compared to [15], because SmarTmem only requires one single allocator, all the tmem is owned by the current node and no inter-node transfer of pages takes place. All of this implies that no special hardware support for SmarTmem is required and can be deployed in any computer server available on the market as long as it provides basic virtualization support, (whereas [15] requires specialized hardware support).

In case there's no free tmem or if the amount of tmem used by the VM is equal or exceeds its target, then any *put* will fail (lines 5–8), forcing the VM to swap to disk. Every *put* issued by a VM will fail as long as the target is equal or smaller to the current amount of tmem in use. The hypervisor can reclaim tmem pages from a VM very slowly, but pages can also be released when a VM explicitly flushes a page (line 16). While the target remains smaller than the tmem capacity in use, the VM will be unable to acquire more tmem pages.

The values of $vm\_data_{\text{hyp}}[id].tmem\_used$ and $vm\_data_{\text{hyp}}[id].puts\_succ$ are incremented when a *put* succeeds (lines 10–13), and a page is allocated for the VM. The parameter $vm\_data_{\text{hyp}}[id].tmem\_used$ is decremented when a VM releases pages (*flush*, lines 16–19), deallocating a tmem

TABLE I: Memory statistics used in SmarTmem. The sampling interval is one second.

Memory Statistics	Description
E_TMEM	Value used in the hypervisor indicating that a put (or other tmem op.) cannot succeed.
S_TMEM	Value used in the hypervisor indicating that a put (or other tmem op.) has succeeded.
$node\_info.free\_tmem$	Number of free pages available for tmem.
$node\_info.vm\_count$	Number of VMs registered.
$vm\_data_{\text{hyp}}[id].vm\_id$	Identifier of the VM within Xen
$vm\_data_{\text{hyp}}[id].tmem\_used$	Number of pages of tmem memory currently used by the VM
$vm\_data_{\text{hyp}}[id].mm\_target$	Target number of pages allocated the VM.
$vm\_data_{\text{hyp}}[id].puts\_total$	Total number of $puts$ issued by the VM in the current sampling interval.
$vm\_data_{\text{hyp}}[id].puts\_succ$	Total number of successful $puts$ issued by the VM in the current sampling interval
$memstats$	Variable storing the last sampled statistics that the hypervisor sent to the MM.
$memstats.vm\_count$	Amount of active VMs as seen by the MM.
$memstats.vm[i].vm\_id$	Identifier of the VM within the MM.
$memstats.vm[i].puts\_total$	Number of $puts$ issued by a VM in the sampling interval.
$memstats.vm[i].puts\_succ$	Number of $puts$ of a VM that succeeded in the current sampling interval.
mm_out	Data structure that holds the output parameters of the MM policy.
$mm\_out[i].vm\_id$	VM identifier that maps a VM to its target allocation as calculated by the MM.
$mm\_out[i].mm\_target$	Memory allocation target as calculated by the policy in the MM.

page. The parameter $vm\_data_{\text{hyp}}[id].puts\_total$ (line 15) is incremented when a *put* occurs, regardless if it succeeds or not.

It is possible for a VM to use more tmem than its target. This can happen because the targets are continuously modified, and the target for a VM might be reduced below the capacity it is using. This might occur because the VM might be temporarily idle, or executing a phase of the application with reduced memory pressure with respect to previous ones. However, this VM won't be able to obtain additional pages until it releases enough pages below its target or until its target is increased.

C. Tmem Kernel Module (TKM)

The TKM is a kernel module that functions as an interface between user-space processes and the tmem implementation of the hypervisor. The TKM provides support for the baseline tmem interface through a series of hypercalls. It also provides additional support for special interrupts generated by the hypervisor to initiate communication with user-space processes, which requires a series of custom-made hypercalls.

The TKM forwards the memory statistics sent by the hypervisor to a user-space process (MM) through a netlink socket interface. As mentioned before, the MM uses these statistics to calculate tmem target allocations, and the TKM forwards this information from the MM back to the hypervisor, for which a series of custom-made hypercalls were also developed.

D. Memory Manager Process for Tmem Allocation in a Virtualized Computing Node

As mentioned in Section III-B, the MM receives information from the hypervisor regarding the way the VMs make use of their memory. The MM keeps track of this information across time, generating a history of how the VMs use tmem, their tmem operations and overall system memory status. The MM uses this information to calculate a tmem capacity target per VM according to custom-made high-level policies.

E. High-Level Tmem Management Policies

Currently, we have implemented three policies in the MM besides the greedy approach used by default (*greedy*), which are: 1) static memory capacity allocation, 2) reconfigurable static capacity allocation, and 3) smart allocation policy. We will proceed to explain and analyze each of the policies.

Algorithm 2 Static Allocation Policy

```
1: function STATIC_POLICY(memstats, node_info)
2:     num_vms ← memstats.vm_count
3:     ms_prev ← memstats.prev
4:     local_tmem ← node_info.total_tmem
5:     mm_target ← local_tmem/num_vms
6:     for i ← 1, num_vms do
7:         mm_out[i].vm_id ← memstats.vm[i].vm_id
8:         mm_out[i].mm_target ← mm_target
9:     end for
10:    send_to_hypervisor(mm_out)
11: end function
```

*1) Static Memory Capacity Allocation (**static-alloc**):* This policy divides the available tmem capacity equally across all tmem-capable VMs, as described by Algorithm 2. This policy secures a fair share of tmem for every VM, since it assumes that each VM has a similar demand for memory.

The MM sends target allocations to the hypervisor when it has calculated a new tmem target. This feature is implemented within the $send\_to\_hypervisor()$ function. If no changes are detected, then no transmission takes place, avoiding unnecessary communication overhead.

For the static allocation policy, the targets are only modified when a new VM is created (and registers itself with tmem) or a VM is destroyed. After that, the targets will remain the same as long as all the VMs stay active, regardless of the applications they are running. This policy is designed to avoid starvation on the available tmem capacity, but it might allocate pages unnecessarily to a VM that does not need to use tmem.

*2) Reconfigurable Static Allocation (**reconf-static**):* This policy divides the available tmem capacity equally among the VMs that are actively using tmem. That is, the policy monitors the activity of each tmem-capable VM, and allocates an equal share of the tmem capacity to each VM that has performed at least one tmem *put*, initially allocating no tmem capacity to any VM. The pseudocode is given in Algorithm 3.

Algorithm 3 Reconfigurable Static Allocation Policy

```
1: function RECONF_STATIC(memstats, node_info)
2:     num_vms ← memstats.vm_count
3:     num_active_vms ← 0
4:     for i ← 1, num_vms do
5:         puts_failed ← memstats.vm[i].cumul_puts_failed
6:         if puts_failed > 0 then
7:             num_active_vms ← num_active_vms + 1
8:         end if
9:     end for
10:    local_tmem ← node_info.total_tmem
11:    for i ← 1, num_vms do
12:        mm_target ← local_tmem/num_active_vms
13:        mm_out[i].vm_id ← memstats.vm[i].vm_id
14:        mm_out[i].mm_target ← mm_target
15:    end for
16:    send_to_hypervisor(mm_out)
17: end function
```

If additional VMs start issuing *puts*, then the tmem capacity allocation is reconfigured, and every VM that *puts* at least once will get an equal amount of the tmem capacity. This will remain so during the lifetime of the VMs. The main drawback of this approach is that it requires for the VM to swap a number of times before getting any tmem capacity, since their initial target allocation is equal to zero. This is because the latency between the time a VM performs its first tmem operation until the time that this condition is detected by the MM and the target are reset, is roughly one second. On the other hand, this policy prevents the hypervisor from allocating memory to a VM unnecessarily as it might occur for *static-alloc*.

*3) Smart Allocation (**smart-alloc**):* This policy assigns tmem capacity to each VM depending on their needs. It monitors every VM, and when it detects that a VM cannot acquire more tmem because it exceeded its target, the policy increases its target by a percentage P of the total *local* tmem capacity, which is constant. It is possible that all VMs have swapped during the last interval, needing a target increase by a percentage P to avoid performance loss. Algorithm 4 shows the pseudo-code for *smart-alloc*.

Algorithm 4 uses the $failed\_puts$ that have occurred in the last sampling interval as a measure of a VM's swap activity, instead of calculating the corresponding rate, as in [15]. The sampling interval is fixed at one second. In this case, it is not necessary to establish communication with other nodes, nor to keep track of node IDs, and the total tmem allocation of the node does not change (as opposed to [15]). Also, *smart-alloc* refrains from sending targets to the hypervisor if they do not change since the last modification, a condition evaluated by the function $send\_to\_hypervisor$ in line 34.

If all the VMs have their allocation increased by a percentage P at every interval, eventually the tmem pages could end up being overallocated i.e. the sum of the targets of every VM ($sum\_targets$, in lines 4 and 25) is larger than the amount of local tmem pages (line 27). If this happens, it is unlikely that the VMs will be able to meet their targets. To avoid this, we make sure that the following condition is met:

$$\sum_{i=0}^{i=n-1} vm\_data_{\text{MM}}[i].mm\_target = local\_tmem \quad (1)$$

Algorithm 4 Smart Allocation Policy

```
1: function SMART-ALLOC_POLICY(memstats, node_info, P)
2:     local_tmem ← node_info.total_tmem
3:     num_vms ← memstats.vm_count
4:     sum_targets ← 0
5:     for i ← 1, num_vms do
6:         put_total ← memstats.vm[i].puts_total
7:         put_succ ← memstats.vm[i].puts_succ
8:         failed_puts ← put_total − put_succ
9:         if failed_puts > 0 then
10:            curr_tgt ← memstats.vm[i].mm_target
11:            incr ← (P × local_tmem)/100
12:            mm_target ← curr_tgt + incr
13:        else
14:            curr_tgt ← memstats.vm[i].mm_target
15:            curr_use ← memstats.vm[i].tmem_used
16:            difference ← curr_tgt − curr_use
17:            if difference > threshold then
18:                mm_target ← ((100 − P) × curr_tgt) /100
19:            else
20:                mm_target ← curr_tgt
21:            end if
22:        end if
23:        mm_out[i].vm_id ← memstats.vm[i].vm_id
24:        mm_out[i].mm_target ← mm_target
25:        sum_targets ← sum_targets + mm_target
26:    end for
27:    if sum_targets > local_tmem then
28:        factor ← local_tmem/sum_targets
29:        for i ← 1, num_vms do
30:            new ← factor × mm_out[i].mm_target
31:            mm_out[i].mm_target ← new
32:        end for
33:    end if
34:    send_to_hypervisor(mm_out)
35: end function
```

When the pages are over-allocated, we reduce the target of every VM according to the following equation:

$$target_{vm_i} = \frac{local\_tmem \times vm\_data_{\text{MM}}[i].mm\_target}{\sum_{i=0}^{i=n-1} vm\_data_{\text{MM}}[i].mm\_target} \quad (2)$$

By enforcing Equations 1 and 2, the policy ensures that: 1) *all local tmem pages* are always assigned to a VM i.e. there are no tmem pages that will remain unallocated, and 2) the total amount of *local tmem pages* that are assigned to the VMs does not exceed the amount of tmem pages available in the node. If these conditions are not enforced, and tmem overallocation occurs, the VMs will compete for the excess pages, overriding the target allocations and unable to meet their targets. Equation 2 is implemented in lines 27–33. It ensures fairness by recalculating the targets and proportionally reallocating tmem.

The policy decreases the target of a VM by a percentage P of the amount of tmem *it has* if the policy detects that a VM is using less pages than its target plus a *threshold* value (lines 16–21). This avoids premature target decrements which might cause the targets to oscillate resulting in an unstable policy.

IV. Benchmarking and Experimental Framework

We tested SmarTmem using nested virtualization. We used a VirtualBox image with Xen 4.5.1, with all VMs running Ubuntu 14.04 with kernel 3.19. The VirtualBox environment was executed with two processor cores, 6 GB of RAM, 2 GB

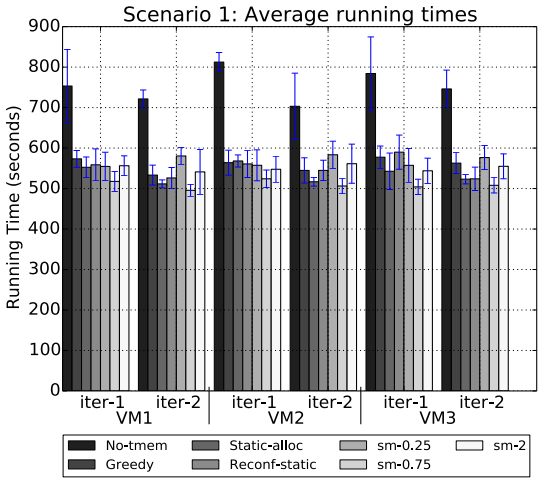

Fig. 3: Running times for Scenario 1. SM refers to *smart-alloc*

of swap and a 32 GB hard drive. The physical system in which VirtualBox was executed had a 4-core Intel Core i7 processor running at 2.1 GHz with 8 GB of RAM, 4 GB of swap and a 320 GB hard drive.

We used the CloudSuite Benchmarks [16] and a micro-benchmark called Usemem to evaluate SmarTmem. Usemem is a synthetic micro-benchmark that allocates an incremental amount of memory as it executes, starting from 128 MB and increasing it by 128 MB increments. Once it allocates a region of memory, it traverses it linearly performing write/read operations. Once it completes a run through a region, it then allocates a larger block, until it reaches 1 GB. Once there, Usemem stops increasing the allocation but continues to write/read on the 1 GB of memory allocated until stopped.

To demonstrate the effectiveness of SmarTmem and the policies, we ran multiple VMs in different execution scenarios, described in Table II. This table shows the scenario names, the VM parameters (CPU and RAM) and a description of the benchmark execution, together with the datasets used. Every scenario is executed five times with every policy. In every scenario, the amount of tmem enable was 1 GB, except for the *Usemem* Scenario in which only 384 MB was enabled.

The VM parameters in each scenario were chosen for the benchmarks to work in a realistic setting in relation to the algorithms and datasets used, so that an enough and reasonable amount of memory pressure is generated during execution. In this way, a fair comparison can be made for all the policies.

V. Results and Discussion

A. Results for Scenario 1

Figure 3 shows the average running times (less is better) and standard deviations for Scenario 1, with the different policies. We vary P for *smart-alloc* and compare the running times against the case without tmem support (*no-tmem*).

Figure 4 shows the amount of tmem capacity of each VM for (a) *greedy* and (b) *smart-alloc* with $P = 0.75\%$. In Figure 4(a), VM3 reaches a tmem capacity peak during the first run, while VM1 and VM2 can't reach their fair share.

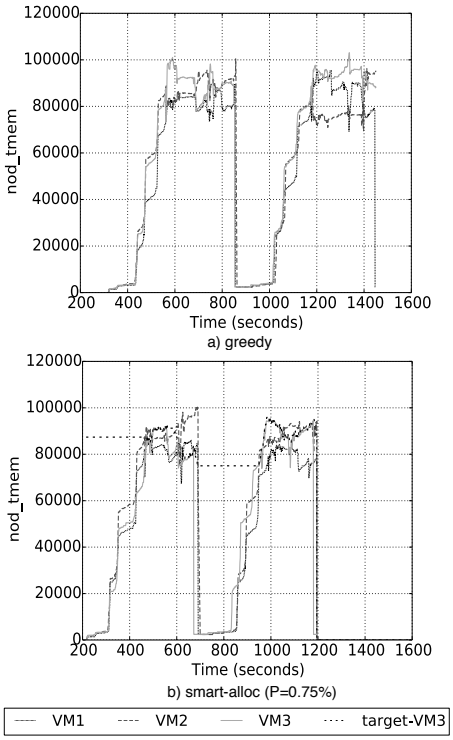

Fig. 4: Utilization of the tmem capacity (nod-tmem) by every VM in number of pages for Scenario 1. The label *target-VM3* refers to the target allocation of the third VM.

During the second run, VM2 is unable to reach a fair share, while the other two VMs take a higher portion of tmem. Figure 4(b) shows that *smart-alloc* maintains similar tmem capacity across VMs with some adaptability, and it also shows the allocation target for VM3 and the way it is enforced.

Some policies were not able to obtain performance benefits. *reconf-static* showed almost no improvement in most of the VMs. Something similar happens to *static-alloc* in the first run of VM2. The case of *smart-alloc* with $P = 0.25\%$ performed poorly for almost every case. The lack of improvement shown by some policies can be attributed to their lack of adaptability to the changes on the memory demand. For example, when using *smart-alloc* with $P = 0.25\%$, the allocation targets increase at a slower pace, causing the VMs to swap more, demonstrating the need for the policy to adapt quickly to the VM's memory demand, for which it is necessary to tune the parameters of *smart-alloc*.

The best runtime performance (fastest) is obtained for *smart-alloc* (referred to as *sm* in Figure 3) with $P = 0.75\%$. Its standard deviation barely overlaps with *greedy*, demonstrating its clear benefits. *smart-alloc* with $P = 0.75\%$ runs faster than *no-tmem* by a maximum of 35.7% (first run of VM3) and by a minimum of 28% (second run of VM2). It also runs faster than *greedy* by a maximum of 12.7%, corresponding to the first run of VM3, and by a minimum of 7.1%. These results demonstrate the need to manage tmem with an adequate management policy.

TABLE II: List of scenarios used for benchmarking. In all cases, we deploy 3 VMs.

Scenario Name	VM Parameters	Comments
Scenario 1	VM1, VM2, VM3: 1 GB RAM, 1 CPU	All VMs execute in-memory-analytics once simultaneously, sleep for 5 seconds and execute it again. The data set was taken from [17].
Scenario 2	VM1, VM2, VM3: 512 MB RAM, 1 CPU	All VMs execute graph-analytics once. The first two VMs launch the benchmarks simultaneously, and the third one launches it 30 seconds later. They all use the same dataset provided by [18], [19], [20].
Usemem Scenario	VM1, VM2, VM3: 512 MB RAM, 1 CPU	All VMs execute *usemem*. VM1 and VM2 start executing *usemem* simultaneously, and VM3 starts when VM1 and VM2 attempt to allocate 640 MB of memory. From this point on, all VMs run concurrently and they are stopped simultaneously when VM3 attempts to allocate 768 MB.
Scenario 3	VM1, VM2: 512 MB RAM, 1 CPU VM3: 1 GB RAM, 1 CPU	VM1 and VM2 execute graph-analytics and VM3 execute in-memory analytics. VM1 and VM2 launch execution simultaneously, and VM3 launches it 30 seconds later. All VMs use the dataset from [18]–[20].

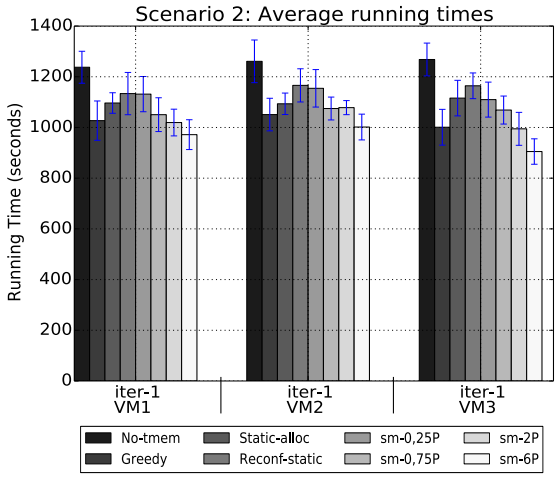

Fig. 5: Running times for Scenario 2.

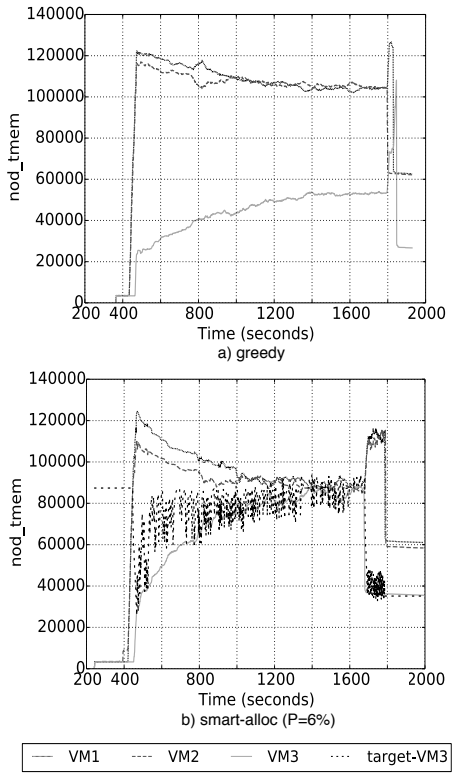

Fig. 6: Tmem use of all VMs in Scenario 2 for a) greedy, and b) *smart-alloc* with $P = 6\%$

B. Results for Scenario 2

The average running times for Scenario 2 are shown in Figure 5 and Figure 6 shows the use of tmem capacity for the VMs. In this case, VM1 and VM2 take a lot of tmem as they start executing since their memory demand rapidly increases, putting significant pressure on the tmem capacity. This is shown in Figures 6(a) and (b), for the case of *greedy* and *smart-alloc* with $P = 6\%$, respectively. In Figure 6(a), the third VM is unable to obtain a fair share of tmem. But in Figure 6(b), when using *smart-alloc*, despite the fact that the first two VMs initially take up a large amount of tmem capacity really fast, the third VM is able to eventually obtain a fair amount. This shows that *smart-alloc* is at the same time adaptive and fair on how it allocates tmem.

In this case, the best performance is obtained with *smart-alloc* with $P = 6\%$, performing better than *no-tmem* by a minimum and a maximum of 21% (for VM3) and 28% (for VM1), respectively. It performs better than *greedy* by minimum and a maximum of 4.7% (for VM2) and 9.6% (for VM3), respectively. This is in contrast to Scenario 1, for which the best benefit for *smart-alloc* was obtained with $P = 0.75\%$. The static policies do not present any improvement in this scenario. These results demonstrate that fairness in tmem allocation among VMs and quick adaptiveness to memory demand spikes are necessary to improve performance. In this scenario, *smart-alloc* is able to achieve both.

C. Results for the Usemem Scenario

The average running times for the *Usemem* Scenario are shown in Figure 7. The *usemem* micro-benchmark is designed to generate a similar memory demand in each VM, giving insight on how the policies behave.

The running times were approximately equal for all VMs when using *static-alloc* for the same amount of memory allocated. Nevertheless, improvements were observed for *reconf-static* when allocating 640 MB for VM2 and consistently across all allocations for VM3. In this case, *static-alloc* policy ensures fairness (not adaptiveness) on how tmem is allocated,

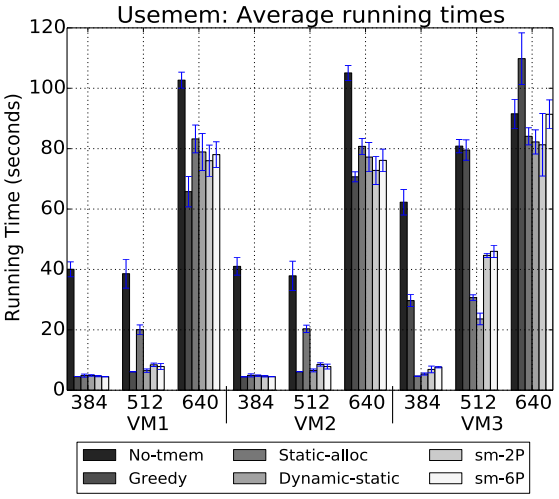

Fig. 7: Running times for *usemem* scenario.

regardless of how much a VM is swapping, and *reconf-static* follows the same behavior depending how many VMs are actually swapping (limited adaptiveness).

Notice how *static-alloc* and *reconf-static* perform worse than *greedy* for VM1 and VM2, but performing better for the third VM across all memory allocations. This is because when VM1 and VM2 are executing, the tmem capacity is not under pressure, but the policies are still enforced restricting the access to tmem unnecessarily. Remarkably, *greedy* performs significantly worse than *no-tmem* for the last allocation of *usemem* in VM3 (640 MB). However, all the other policies perform better than *no-tmem* for VM3.

Figure 8(a) shows that VM3 struggles to obtain tmem pages when using *greedy* as the tmem capacity is under pressure. This is similar to the case presented in Figure 6(a) for Scenario 2. However, in Figure 8(b) and Figure 8(c), corresponding to *reconf-static* and *smart-alloc* with $P = 2\%$, every VM is able to obtain a fair share of memory.

The case of *smart-alloc* allows for the VM1 and VM2 to take much more memory than with *reconf-static*, but still less than what *greedy* allows, as seen on Figures 8(a) and 8(c). In this case, *smart-alloc* exhibits more *adaptiveness*. *reconf-static* performs better for when VM3 allocates 384 MB and 512 MB, as previously seen. These results demonstrate that *reconf-static* and *static-alloc*, both of which are more oriented towards fairness, despite not showing the best performance in Scenarios 1 and 2, still present better performance than *smart-alloc* for these *Usemem* cases.

When the third VM starts executing *Usemem*, *smart-alloc* achieves fair allocation at a slower pace. In Figure 8(c), the three VMs take more tmem than the limit imposed by *reconf-static*, demonstrating the better adaptiveness of *smart-alloc*.

These results are interesting because they highlight a trade-off between the *adaptiveness* of a policy to quickly respond to changes in memory demand (as *smart-alloc*) and its *ability* to fairly allocate tmem (as *static-alloc* and *reconf-static*). The more responsive a policy is, the more tmem it will let a VM take, but as other VMs come under pressure, it will be harder to reach fairness.

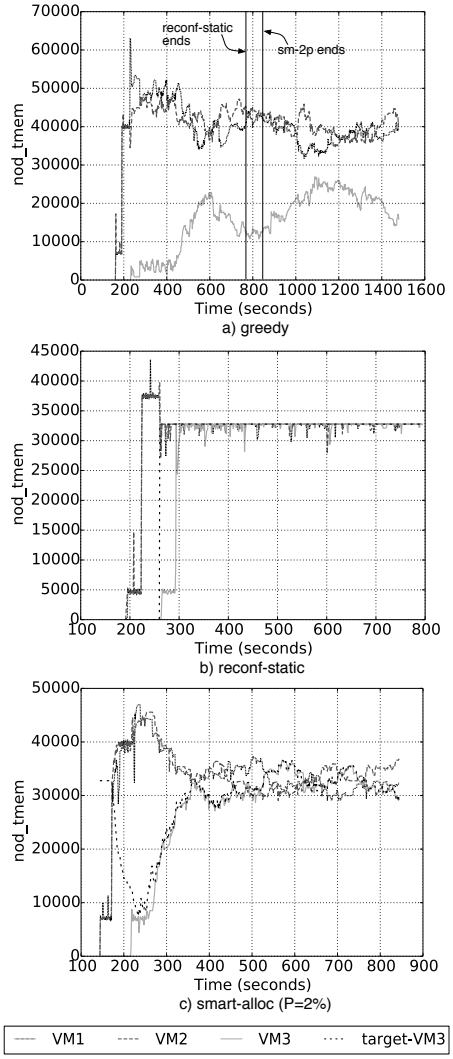

Fig. 8: Tmem use of all VMs in *usemem* for a) greedy, b) *reconf-static* and c) *smart-alloc* with $P = 2\%$. The vertical lines in a) show the time when *usemem* completes execution.

D. Results for Scenario 3

The average running times for Scenario 3 are shown in Figure 9, and Figure 10 show the tmem capacity used by the three VMs for *greedy*, *static-alloc*, *reconf-static* and *smart-alloc* with $P = 4\%$.

The *graph-analytics* benchmark starts by making use of a large amount of tmem. For *greedy*, VM1 and VM2 take up half of the available memory each, leaving almost no memory available for when VM3 increases its demand. This helps explain why VM3 executes very slow with *greedy*. Notice the way *static-alloc* is very rigid (not adaptive), setting an upper-bound for all three VMs. This seems to benefit VM3 very much and, *surprisingly*, VM1 and VM2 also improved.

In Figure 10(c), we see the case with *reconf-static*. This policy allows for VM1 and VM2 to share half of the available

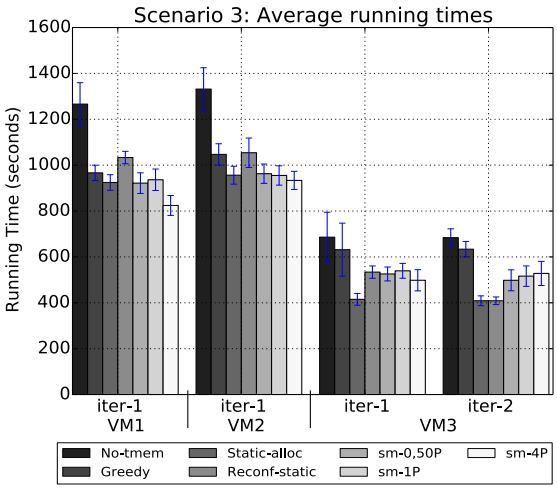

Fig. 9: Running times for Scenario 3.

capacity until VM3 starts swapping. By this time, the targets are reconfigured but VM3 is still unable to reach a fair share. because pages are released by the VM at a slower pace, despite the targets already being modified.

Notice how *smart-alloc* allows for VM1 and VM2 to take up a similar amount of tmem when compared to *greedy*, but reduces their share as soon as VM3 swaps, but VM3 is unable to reach a fair share, allowing VM1 and VM2 to have more tmem. This helps explain the results observed in Figure 9, where VM1 and VM2 run faster for *smart-alloc* but slower for *static-alloc*, while it is the opposite for VM3. Again, this highlights the trade-off between adaptiveness and fairness.

All the policies improve over *greedy* consistently, except for *reconf-static* which fails for the first two VMs. For VM1 and VM2, *smart-alloc* with $P = 4\%$ performs better than the other policies. For VM3, the best performance is obtained with *static-alloc* by a very significant margin. The maximum and minimum improvements over *no-tmem* are 40% (in VM3 for *static-alloc*) and 22% (in VM for *smart-alloc* with $P = 4\%$), respectively, while the maximum and mininum improvements over *greedy* are 35% (in VM3 for *static-alloc*) and 10.8% (in VM2 for *smart-alloc* with $P = 4\%$), respectively.

VI. Related Work

Many research efforts seek to optimize memory allocation by implementing better control strategies in the balloon driver [10], [21]–[25]. One similarity we have with respect to Liu et al. [10] is that we also implement a user-space process to manage the allocation of memory capacity to every VM, albeit their communication mechanisms are slightly more complicated, since they need to communicate the allocation targets to every VM. In our case, it is the hypervisor in charge of doing so, which reduces significantly the memory and communication overheads.

Both Zhao et al. [22] and Liu et al. [10] predict the memory demands of the VMs. Whereas, Zhao et al. [22] predicts the working set size of the VMs to dictate target memory allocations to the balloon driver, Liu et al. [10] predicts the

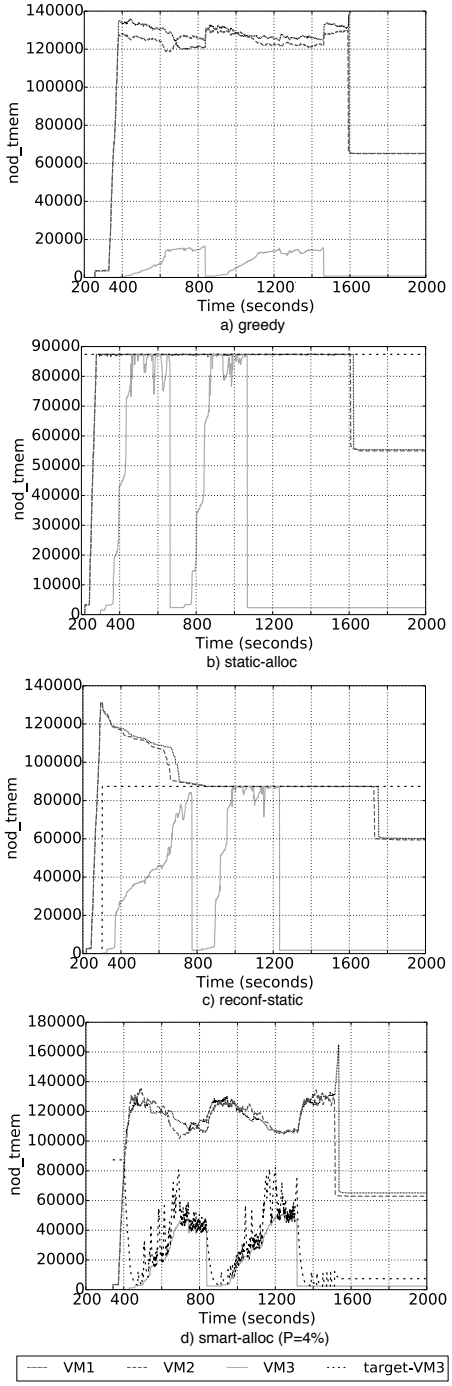

Fig. 10: Tmem use of all VMs in Scenario 3 for a) *greedy*, b) *static-alloc*, c) *reconf-static* and d) *smart-alloc* with $P = 4\%$.

total memory footprint including swap. In contrast, rather than predicting memory requirements, we shift the core of the problem into directly reducing the amount of swapping to disk, thus keeping the data in system memory. In addition, we focus entirely on tmem instead of memory ballooning.

Venkatesan et al. [26] use tmem, in both cleancache and frontswap mode, in a computer node with non-volatile memory (NVM) alongside traditional DRAM. They use tmem to access

the NVM where the disk device would have normally been used. They divide the NVM space into a clean region (for clean cache pages) and a swap region (for processes that attempt to swap to disk). One difference between SmarTmem and Venkatesan et al. [26] is that we only make use of tmem on its frontswap mode. The type of applications we run from Cloudsuite are memory intensive, and the processes of these applications dynamically allocate memory pages that are not backed by the filesystem. Thus, when data is evicted from memory, those pages have to be stored in frontswap.

VII. Conclusions and Future Work

This paper introduces a mechanism, based on tmem, which exploits memory resources in a virtualized computing node executing multiple applications. Our results demonstrate the need to intelligently and efficiently allocate tmem capacity to VMs at the node level, showing significant performance improvements, of up to 35%, over the default greedy policy. At the same time, we identified a trade-off between the policy's adaptiveness and its fairness. These results will be relevant also for tmem-based memory disaggregation solutions and to systems that use the tmem interface with heterogenous memory architectures such as Venkatesan et al. [26].

We evaluated the effectiveness of SmarTmem, in terms of fairness among VMs and adaptiveness to changing memory demands. This evaluation was done using micro-benchmarks and application scenarios using CloudSuite benchmarks. We also demonstrated that the policies implemented ensure fair capacity allocation in different scenarios, while being able to adapt to the changing memory demand of the VMs. This paper provides a framework and baseline for future development of more sophisticated tmem memory policies, as well as integration of tmem and other memory allocation mechanisms.

Acknowledgements

This research is part of a project that has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 754337 (EuroEXA) and the European Unions 7th Framework Programme under grant agreement number 610456 (Euroserver). It also received funding from the Spanish Ministry of Science and Technology (project TIN2015-65316-P), Generalitat de Catalunya (contract 2014-SGR-1272), and the Severo Ochoa Programme (SEV-2015-0493) of the Spanish Government.

References

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *Commun. ACM*, vol. 53, no. 4, April 2010.

[2] Q. Zhang, L. Cheng, and R. Boutaba, "Cloud computing: state-of-the-art and research challenges," *Journal of Internet Services and Applications*, vol. 1, no. 1, pp. 7–18, 2010.

[3] Cloudstack, "https://cloudstack.apache.org/."

[4] Openstack, "http://www.openstack.org/."

[5] Z. Zhang, C. Wu, and D. W. Cheung, "A survey on cloud interoperability: Taxonomies, standards, and practice," *SIGMETRICS Perform. Eval. Rev.*, vol. 40, no. 4, pp. 13–22, April 2013.

[6] C. Gong, J. Liu, Q. Zhang, H. Chen, and Z. Gong, "The characteristics of cloud computing," in *Proceedings of the 2010 39th International Conference on Parallel Processing Workshops*, ser. ICPPW '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 275–279.

[7] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *Commun. ACM*, vol. 17, no. 7, pp. 412–421, July 1974.

[8] J. Sahoo, S. Mohapatra, and R. Lath, "Virtualization: A survey on concepts, taxonomy and associated security issues," in *Proceedings of the 2010 Second International Conference on Computer and Network Technology*, ser. ICCNT '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 222–226.

[9] J. Simão, J. Singer, and L. Veiga, "A comparative look at adaptive memory management in virtual machines," in *Proceedings of the 2013 IEEE International Conference on Cloud Computing Technology and Science - Volume 01*, ser. CLOUDCOM '13, vol. 1. Washington, DC, USA: IEEE Computer Society, 2013, pp. 452–457.

[10] H. Liu, H. Jin, X. Liao, W. Deng, B. He, and C.-z. Xu, "Hotplug or ballooning: A comparative study on dynamic memory management techniques for virtual machines," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 5, pp. 1350–1363, 2015.

[11] C. A. Waldspurger, "Memory resource management in vmware esx server," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 181–194, December 2002.

[12] K. Fraser and M. J. Silbermann, "Resizing memory with balloons and hotplug," in *In Proceedings of the Linux Symposium*, 2006, pp. 313–319.

[13] D. Magenheimer, C. Mason, D. McCracken, and K. Hackel, "Transcendent memory and linux," in *Ottawa Linux Symposium*, July 2009, pp. 191–200.

[14] X. Li, P. Zhang, R. Chu, and H. Wang, "Optimizing guest swapping using elastic and transparent memory provisioning on virtualization platform," *Front. Comput. Sci.*, vol. 10, no. 5, pp. 908–924, October 2016.

[15] L. Garrido and P. Carpenter, "vmca: Memory capacity aggregation and management in cloud environments," in *IEEE 23rd Intl. Conference on Parallel and Distributed Systems (ICPADS)*, December 2017.

[16] 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 *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '12. New York, NY, USA: ACM, 2012, pp. 37–48. [Online]. Available: http://doi.acm.org/10.1145/2150976.2150982

[17] F. M. Harper and J. A. Konstan, "The movielens datasets: History and context," *ACM Trans. Interact. Intell. Syst.*, vol. 5, no. 4, pp. 19:1–19:19, Dec. 2015. [Online]. Available: http://doi.acm.org/10.1145/2827872

[18] Rossi, Ryan A. and Ahmed, Nesreen K., "soc-twitter-follows - social networks," http://networkrepository.com/soc-twitter-follows.php, 2013.

[19] Rossi, R. A. and Ahmed, Nesreen K., "The network data repository with interactive graph analytics and visualization," in *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

[20] Rossi, R. A. and Ahmed, N. K., "An interactive data repository with visual analytics," *SIGKDD Explor.*, vol. 17, no. 2, pp. 37–41, 2016.

[21] 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*, ser. EuroSys '13. New York, NY, USA: ACM, 2013, pp. 337–350.

[22] W. Zhao, Z. Wang, and Y. Luo, "Dynamic memory balancing for virtual machines," in *SIGOPS Oper. Syst. Rev*, vol. 43, no. 3. ACM, July 2009, pp. 37–47.

[23] J. Kim, V. Fedorov, P. V. Gratz, and A. L. N. Reddy, "Dynamic memory pressure aware ballooning," in *Proceedings of the 2015 International Symposium on Memory Systems*, ser. MEMSYS '15. New York, NY, USA: ACM, 2015, pp. 103–112.

[24] J.-H. Chiang, H.-L. Li, and T. cker Chiueh, "Working set-based physical memory ballooning," in *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*. San Jose, CA: USENIX, 2013, pp. 95–99.

[25] G. Moltó, M. Caballer, E. Romero, and C. de Alfonso, "Elastic memory management of virtualized infrastructures for applications with dynamic memory requirements," in *International Conference on Computational Science*, vol. 18, no. 159-168, 2013.

[26] V. Venkatesan, W. Qingsong, and Y. C. Tay, "Ex-tmem: Extending transcendent memory with non-volatile memory for virtual machines," in *Proceedings of the 2014 IEEE Intl Conf on High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC,CSS,ICESS)*, ser. HPCC '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 966–973.