vMCA: Memory Capacity Aggregation and Management in Cloud Environments

Luis A. Garrido
Barcelona Supercomputing Center
Barcelona, Spain
Email: luis.garrido@bsc.es

Paul Carpenter
Barcelona Supercomputing Center
Barcelona, Spain
Email: paul.carpenter@bsc.es

***Abstract*—In cloud environments, the VMs within the computing nodes generate varying memory demand profiles. When memory utilization reaches its limits due to this, costly (virtual) disk accesses and/or VM migrations can occur. Since some nodes might have idle memory, some costly operations could be avoided by making the idle memory available to the nodes that need it. In view of this, new architectures have been introduced that provide hardware support for a shared global address space that, together with fast interconnects, can share resources across nodes. Thus, memory becomes a global resource.**

This paper presents a memory capacity aggregation mechanism for cloud environments called vMCA (Virtualized Memory Capacity Aggregation) based on Xen's Transcendent Memory (Tmem). vMCA distributes the system's total memory within a single node and globally across multiple nodes using a user-space process with high-level memory management policies. We evaluate vMCA using CloudSuite 3.0 on Linux and Xen. Our results demonstrate a peak running time improvement of 76.8% when aggregating memory, and of 37.5% when aggregating memory and implementing our policies.

***Keywords*-Virtualization; Memory; Aggregation; Tmem; vMCA**

I. Introduction

Cloud data center environments are built using "share nothing" servers, each provisioned with their own computing resources, communicating over a TCP/IP (or similar) network. New system architectures [5, 7, 8], however, have been proposed that present a shared global physical address space and use a fast interconnect to share physical resources through the memory hierarchy. An example is the UNIMEM (*Unified Memory*) memory model, which is an important feature of the "EuroEXA" family of projects [7, 12]. In such systems, remote memory is addressable at low latency using Remote DMA and/or load/store instructions. Cache coherency is only enforced inside a node (a *coherence island*), which avoids global coherence traffic and enables scalability to large numbers of nodes.

Such architectures present an opportunity to aggregate physical memory capacity. In cloud environments, memory is currently managed at the node level, with the hypervisor distributing this capacity among its Virtual Machines (VMs), each of which hosts a guest OS. The hypervisor usually overcommits its memory, which allows the node to have more active VMs by assuming they won't use their full allocation. This improves the utilization of the node's memory but also increases pressure on it, even more so if the VMs have varying demands for memory. To alleviate this pressure, dynamic memory management mechanisms are implemented, such as memory ballooning and hotplug. Xen's Transcendent Memory (Tmem) [1] is another way to make additional memory capacity available to the VMs, which works by anticipatedly pooling all underutilized memory capacity.

This paper presents vMCA (virtualized Memory Capacity Aggregation), an extension of Xen's Tmem that leverages virtualization software and a global address space to allow the whole system's memory capacity to be shared among nodes. vMCA extends and improves upon an earlier proposal known as GV–Tmem [9], in order to handle resiliency, and with a deeper discussion and improvements of the memory aggregation and allocation policies. Like GV–Tmem, vMCA introduces minimal changes to the hypervisor to enforce constraints on local and remote page allocation. The complexity is situated in a user-space Memory Manager (MM) in the privileged domain of the nodes, which implements a resilient and reliable mechanism for distributing memory capacity across nodes according to a high-level memory management policy. The main contributions of this paper are:

1) A software stack to aggregate memory capacity across multiple nodes, focused on resiliency and efficiency.
2) A multi-level user-space mechanism for allocation/management of aggregated memory capacity, with special considerations for memory allocation within a node and memory distribution across nodes.
3) A complete analysis of high-level policies for memory aggregation and allocation.

This paper is organized as follows. Section II provides the necessary background. Section III presents vMCA and its components, explaining its resiliency to failures and the necessary hardware support. Section IV describes the experimental methodology and Sect. V shows the evaluation results. Section VI compares with related work. Finally, Section VII outlines future work and concludes the paper.

II. Background

This section gives an overview on virtualization and memory management in cloud data centers, on Xen's tran-

scendent memory and outlines the hardware architectures which implement coherence islands.

A. Virtualization Technology in the Cloud

There are many cloud service models available in the industry, such as SaaS (Software-as-a-Service), PaaS (Platform-as-a-Service) and IaaS (Infrastucture-as-a-Service). Considered one of the most important, IaaS allows users to dynamically access a configurable (seemingly unlimited) pool of computing resources. These resources are made available to the VMs using node-level virtualization software known as a hypervisor or Virtual Machine Manager (VMM), e.g. Xen [2], KVM [3] or VMWare [4].

1) Hypervisor and Dynamic Memory Management: The hypervisor virtualizes the computing resources of the node, and creates and manages VMs with their own (guest) OSs. When a VM is created, the hypervisor allocates for it a portion of the physical memory. If the portion allocated is less than the amount of memory the VM needs at some point, then the VM is *under-provisioned* of memory. But when the VM has excess memory, the VM is *over-provisioned*. To optimize the utilization of memory and increase performance, memory has to be re-allocated from the VM that has an excess of memory to the VM that needs it.

Hypervisors implement mechanisms to dynamically re-allocate memory among VMs, including memory ballooning and memory hotplug. These mechanisms have been widely deployed in data centers, obtaining high levels of memory utilization. However, they do not provide adequate interfaces to aggregate memory capacity [22].

2) State-of-the-Art Transcendent Memory: Transcendent memory (Tmem) [1] was introduced as an additional solution for dynamic memory management. It pools the node's physical pages that are under-utilized and/or that are unassigned (fallow) to any VM. Tmem is abstracted as a key–value store in which pages are allocated/written and read through a paravirtualized put–get interface. Data written to Tmem is either *ephemeral* (or *non-persistent*) (e.g. clean data that the hypervisor can discard to reclaim memory) or *permanent* (or *persistent*) (e.g. dirty data that must be maintained) and either *private* (to a VM) or *shared* (among VMs).

Linux can take advantage of Tmem in two ways: *cleancache* and *frontswap*, which have been integrated into the Linux kernel. Both require a special Tmem kernel module to be loaded in order to access the hypervisor's Tmem functionality using the relevant hypercalls. Linux cleancache is a cache for clean pages that are evicted by the Linux kernel's Pageframe Replacement Algorithm (PFRA). Linux frontswap uses Tmem as a swap device cache. When a page is getting swapped, frontswap attempts to store it (put) into Tmem, and it is swapped in case the store attempt fails. Whenever the VM tries to access a page from the filesystem or swap space, it will check whether it was previously put into Tmem. If so, it will get it from Tmem assuming the page wasn't reclaimed. A successful put will avoid the disk write and read. Pages can also be de-allocated using flush-page operations, freeing the page to be used by other VM.

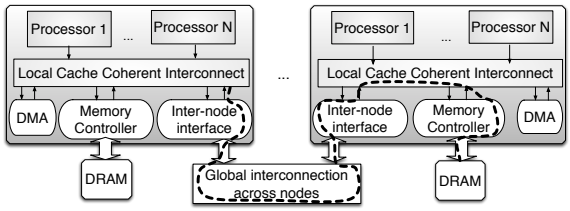

Figure 1. UNIMEM architecture with two coherence islands

B. Hardware Support for Coherence Islands

Recent advances in computer architecture have proposed architectures with a shared global physical address space and no global hardware cache coherence [5–8, 10]. The essential features of architectures like EUROSERVER [7] are captured by the UNIMEM memory model.

UNIMEM is well suited for distributed systems that are able to share resources. Figure 1 illustrates how UNIMEM works. The number of nodes in current UNIMEM-based architectures ranges from 4 to 8 nodes. Each node contains N processor cores (6 to 8 in current implementations), grouped in clusters connected via a local cache-coherent interconnect to local DRAM and I/O devices. Remote memory is visible through the global physical address space, and the *inter-node interface* together with the *global interconnect* establish communication among the nodes, routing the remote memory accesses to the appropriate node. The essential characteristics of these architectures are summarized as:

- A global physical memory address space provides RDMA and load/store access to memory in all nodes.
- Routing uses the top bits of the global physical address.
- Fast, low-latency communication among the nodes.
- Each node executes its own hypervisor and OSs.
- Features required for resiliency (see Section III-C4).

III. vMCA Design

The vMCA stack consists of three components:

- Hypervisor support (Sect. III-A)
- Tmem Kernel Module (TKM) in Dom0 (Sect. III-B)
- Memory Manager (MM) in Dom0 (Sect. III-C)

A. Hypervisor Support

The Xen hypervisor has been extended to support vMCA, and all extensions are localized in the Tmem subsystem, which originally supports the necessary features to allocate and deallocate local Tmem pages on behalf of the VMs.

Page ownership: Each hypervisor *owns* a subset of the physical pages in the system, which constitutes the pool of memory that it can use and allocate to its VMs. When a hypervisor boots and first joins the vMCA system, it owns all of its *local* physical memory; i.e. all pages whose *home* node

Memory Statistics	Description
$ENOMEM$	Code used in the hypervisor to signify that a *put* failed due to a lack of Tmem capacity
$node\_info$	Data structure that holds general status information of the computer host.
$node\_info.total\_tmem$	Total number of pages available to Tmem (free or allocated)
$node\_info.id$	Identifier of the node inside vMCA-supported system
vm_data_{hyp}	Data structure that holds the parameters of all of the VMs within the hypervisor
$vm\_data_{\text{hyp}}[id].vm\_id$	Identifier of the VM within Xen
$vm\_data_{\text{hyp}}[id].tmem\_used$	Number of Tmem pages currently used by the VM
$vm\_data_{\text{hyp}}[id].mm\_target$	Target number of pages for the VM, held by the hypervisor and previously sent by the MM
$vm\_data_{\text{hyp}}[id].puts\_total\_rate$	Total number of $puts$ issued by the VM in the most recent period
$vm\_data_{\text{hyp}}[id].puts\_succ\_rate$	Total number of successful $puts$ issued by the VM in the most recent period
$memstats$	Data structure that holds the last sampled statistics that the hypervisor sent to the MM.
$memstats.vm\_count$	Amount of active VMs as seen by the MM.
$memstats.vm$	Array where each entry holds statistics about an active VM
$memstats.vm[i].vm\_id$	Identifier of the VM within the MM, as received from the hypervisor
$memstats.vm[i].puts\_total\_rate$	Total number of $puts$ that a VM has issued to the hypervisor in the recent period
$memstats.vm[i].puts\_succ\_rate$	Total number of successful $puts$ that a VM has issued to the hypervisor in the recent period
$memstats.total\_alloc\_pages$	Total number of pages available in every zoned buddy allocator of the hypervisor
mm_out	Pointer to a 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

Table I. SUMMARY OF MEMORY STATISTICS USED IN THE MEMORY MANAGER (MM)

Algorithm 1 Enforce memory allocation in the Hypervisor

```
1:  function DO_TMEM_PUT(vm_data_hyp, id)
2:      tmem_used ← vm_data_hyp[id].tmem_used
3:      mm_target ← vm_data_hyp[id].mm_target
4:      if tmem_used ≥ mm_target then
5:          return_value ← −ENOMEM
6:      else if node_info.free_tmem == 0 then
7:          return_value ← −ENOMEM
8:      else
9:          vm_data_hyp[id].tmem_used += 1
10:         vm_data_hyp[id].puts_succ += 1
11:         return_value ← 1
12:     vm_data_hyp[id].puts_total += 1
13:     return return_value
```

is that node. If it needs additional memory, it may request ownership of additional *remote* memory pages. Conversely, it may release ownership of some of its local or remote pages, so that they can be granted to another node that has a greater need for them. As described in Sect. III-C, the MMs collectively ensure that each physical page in the system is owned by at most one hypervisor. The only times when a page is not owned by any hypervisor are when the page is in transit between hypervisors, and, rarely, following a memory leak caused by a failed node (Sect. III-C4).

Page allocation: The pages owned by a hypervisor are allocated using a zoned Buddy allocator, with a zone for each node, including itself, from which it has ownership of at least one page. The allocators should be stored in a tree corresponding to the hierarchy of the system (nodes in leaves, then, e.g. boards, chasses and racks). A Tmem put traverses the tree to find the closest non-empty allocator, and allocates a page from it. A Tmem flush operation frees a page, returning it to the corresponding Buddy allocator.

Transfer of page ownership: In order to minimize communication overheads, page ownership is transferred with hypercalls in *blocks*, each an appropriately-aligned power-of-two number of pages. The hypervisor supports four hypercalls in order to transfer page ownership. The Grant hypercall is used to receive ownership of a list of blocks, which are added to the appropriate Buddy allocator. In contrast, a Request hypercall requests that the hypervisor releases ownership of a number of pages on behalf of another node. In response, the hypervisor takes the necessary blocks from its allocators using a simple heuristic (see **Page allocation**) to prefer large blocks located physically close to the target node. The Return hypercall is issued when another node wishes to shut down or leave vMCA. It asks the hypervisor to release ownership of all pages that are physically located on that node. It returns free pages, and, executing asynchronously, searches for pages in use by the Tmem clients and migrates their contents to free pages. Finally, the Invalidate-Xen hypercall is used if a remote node fails: it discards all free or allocated pages located in that node, and terminates all VMs that were using that data.

Enforcing local per-VM memory constraints: The hypervisor enforces the Tmem allocations determined by the MM. Algorithm 1 illustrates this mechanism. The DO_TMEM_PUT function is called when a VM attempts to store a page in Tmem. The hypervisor checks the amount of Tmem the VM has (line 4), and returns $-ENOMEM$, denying the request, if the VM is already at the limits of its allocation. If not (lines 9–11), the Tmem page will be given to the VM and the necessary data structures are updated.

Tmem Statistics: Table I presents the Tmem statistics gathered by the hypervisor. The hypervisors sends statistics to the MM every second, and their size is minimized to prevent large communication overheads.

Command	Direction	Description	Slave state
Distribution of global memory capacity			
Statistics(S)	S→M	Send node statistics S to Master	Active
Grant-Any(n, x)	S→M	Request n pages to slave x	Active
Grant-Fwd(n, x, y)	M→S	Forward request of n pages coming from y to slave x	Active
Force-Return(n, x)	M→S	Disable node and return all pages located at slave x (for leaving)	Active
Flow of page ownership			
Grant($b, \cdots$)	S/M→S	Transfer ownership of blocks of pages	Active
Node state changes			
Register	S→M	Register a new node	Inactive→Active
Leave-Req	S→M	Node requests to leave the system (e.g. for shutdown)	Active→Leaving
Leave-Notify	M→S	MM–M notifies that the recipient has left the system	Leaving→Inactive
Enable-Node(x, e)	M→S	Inform slave to accept ($e = 1$) or reject ($e = 0$) pages at node x	Active
Resiliency support			
Invalidate	S→M	Invalidate all pages located at sending node	Active/Leaving→Recovery
Invalidate(x)	M→S	Request recipient to invalidate pages at node x	Active
Invalidate-Notify(x)	M→S	MM–M notifies that the recipient has been invalidated	Recovery→Inactive

Table II. MEMORY MANAGER MESSAGE TYPES. IN THE TABLE, "S" STANDS FOR SLAVE AND "M" FOR MASTER

B. Dom0 Tmem Kernel Module (TKM)

The Tmem client interface requires a kernel module in each guest domain. vMCA needs a kernel module in the privileged domain Dom0 that acts as an interface between the hypervisor (using hypercalls) and the node's MM.

C. Dom0 User-space MM

Each node has a user-space MM in its privileged domain, Dom0. The MMs cooperate to:

- Distribute memory owned by each node among its VMs
- Distribute global memory capacity among nodes
- Implement the flow of page ownership among nodes
- Enable nodes to join/leave vMCA, and handle failures

In the current design, one of the MMs is designated to be the Memory Manager Master (MM–M), which is responsible for system control and global memory capacity distribution. The MMs communicate using a secure and reliable packet transport such as SSL/TLS. The commands passed among the MMs are listed in Table II.

1) Joining the vMCA system: In order to join vMCA, a node requires a configuration file, which provides the network address of all the nodes, security credentials for secure connections, and the mapping for all nodes from node ID to network address. For managing locality, it also needs to know the location of each node in the NUMA hierarchy.

When a node R wishes to join vMCA, it first sends a message with a Register command to the MM–M (Table II). The MM–M sets its state to Active (Figure 2) and sends an Enable-Node(R,1) command to all the registered nodes. Each node maintains a bitmap of the enabled nodes.

2) Distributing ownership of memory: The flow of memory pages across vMCA considers many aspects.

Distributing memory owned by a node among guests: Each node's MM determines the maximum Tmem allocation for each VM, using a policy (Sect. III-D) that uses the Tmem statistics gathered by the hypervisor. Pages are dynamically allocated to every VM subject to total (local plus remote) consumption limits, which are sent to the hypervisor using a Mem-Limit hypercall.

For every policy, the following condition must be met:

$$\sum_{i=1}^{i=m} vm\_data_{\text{MM}}[i].mm\_target = total\_tmem(t_i) \quad (1)$$

Eq. 1 means that at time t_i, the amount of total Tmem pages available to the node ($total\_tmem$) is equal to the sum of the allocations of all m active VMs. This implies that every Tmem page is usable, and that Tmem pages are not overallocated. However, when calculating the allocation limits for every VM, it is possible that the sum of the allocations exceeds $total\_tmem(t_i)$.

Overallocation of pages causes VMs to compete for the available Tmem capacity, defeating the purpose of the management policies. When over-allocation occurs, the MM recalculates the target of the VMs as follows:

$$tgt_{vm_i} = \frac{total\_tmem \times vm\_data_{\text{MM}}[i].mm\_target}{\sum_{i=1}^{i=m} vm\_data_{\text{MM}}[i].mm\_target} \quad (2)$$

Eq. 2, where tgt_{vm_i} is an abbreviated form of $vm\_data_{\text{MM}}[i].mm\_target$, ensures that the proportional allocation of each VM follows the policy's desired proportion while also satisfying Eq. 1.

Distributing global memory capacity among nodes: All nodes in the Active state send statistics to the MM–M using the Statistics command. A node that needs memory sends a Grant-Any command to the MM–M, which redistributes the memory based on the statistics and the global memory policy. The MM–M then sends Grant-Fwd commands, forwarding requests to a donor node to transfer ownership of a number of free physical pages to a requesting node.

Flow of page ownership: The MM–M rebalances memory capacity without knowing the physical addresses. Ownership of global physical addresses is transferred peer-to-peer using Grant commands. A Grant command passes

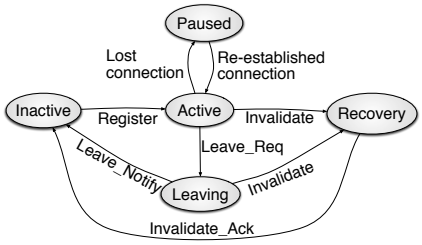

Figure 2. Node state transition diagram (at MM–M)

a list of blocks, each an appropriately-aligned power-of-two number of physical pages. The MM–M can also satisfy the request for memory of another node, for which it will send a Grant command to the requesting node.

To avoid race conditions during node shutdown/failure, the recipient checks the home of each received block against the bitmap of enabled nodes (Sect. III-C1). If the home node of a block is disabled (rarely occurs), then ownership is returned back to its home using a new point-to-point Grant.

3) Leaving the vMCA system: Shutting down a node of vMCA requires the following procedure. Note that if a node fails, then the procedure in Sect. III-C4 is followed.

1) The node R that wishes to shutdown sends Leave-Req to the MM–M.
2) The MM–M in response moves the node to the Leaving state (see Figure 2). It sends a Force-Return(R) command to all nodes in the system. Each recipient returns all of the pages at R that it owns and will reject any such pages received in future Grant commands.
3) Node R frees all pages used by Tmem and returns ownership of all remote pages to their home nodes using peer-to-peer Grant messages.
4) Each node sends a Grant command to node R to return ownership of the pages that it borrowed.
5) The MM–M is disabled once it has received Statistics from every node indicating that R and that it owns no pages at R, then the MM–M moves R to Inactive (Figure 2) and sends Leave-Notify to R.
6) Node R at this point has ownership of all its non-leaked pages, has left vMCA, and may shutdown.

4) Resilient memory capacity aggregation: There are two aspects to ensuring resiliency in the face of node failures: 1) protecting the integrity of the data stored in Tmem, and 2) restoring lost system state.

Ensuring data integrity on slave failure: The Tmem interface guarantees that a get returns the data previously written by the corresponding put. In addition, if the pool is *persistent*, the get operation must succeed.

Assume that node A has ownership of memory located at node R. If R fails, then this data is lost. If R reboots, it will restart and may even attempt to re-join vMCA. Since any get performed at A must never access the (incorrect) new contents of the physical page at node R, the following procedure must be followed after a failure:

1) When a node R boots, remote memory accesses to it must raise a hardware exception (Sect. III-E), also raised when a node accesses a page from a failed node. If a remote access causes such an exception, the VM must be shutdown if the page is in a *persistent* pool. If the pool is *non-persistent*, the get fails.
2) When node R is initialized without a previous clean shutdown, it sends Invalidate to the MM–M.
3) The MM–M moves the node to the Recovery state (Figure 2) and sends Invalidate(R) to all nodes. Each node then disables node R, and if it has ownership of pages with home R, it makes an Invalidate-Xen hypercall to invalidate the pages.
4) Once the MM–M has received Statistics from every node, i.e. it has disabled R and owns no pages at R, then the MM–M moves the node to the Inactive state and sends an Invalidate-Notify command to R.
5) Node R begins its normal initialization procedure upon receiving the Invalidate-Notify command.

Restoring lost pages on slave failure: After a node R fails, vMCA is unable to recover the pages that it owned. It is possible to approximate the number of lost pages from a home node using the statistics sent by the nodes to the MM–M, by adding the number of pages owned and comparing against the number of physical pages at the node. However, since this is done using potentially outdated information from Statistics commands, the value will not be exact.

Restoring system state on MM–M failure: If the MM–M fails, the system will continue to work except that a) no memory capacity will be redistributed, and b) nodes can't enter/leave the system.

When the MM–M attempts to restart, it will listen for connections from the other nodes. When a node connects and starts sending Statistics, it is added to the system in the Active state. A node that is in the process of leaving or invalidation will send a new Leave-Req or Invalidate command on re-connecting to the MM–M, which will restart the leaving or invalidation process. At any time, all nodes in the Active state are part of the global memory allocation.

D. Memory Management Policies

We tested three memory management policies: 1) greedy-local [9] (single-node Tmem implementation), 2) greedy-remote [9] and a 3) two-level memory management policy (TLP), divided in a first and a second level of memory management, implemented for this work. greedy-remote, like greedy-local, allows for the node's hypervisor to give Tmem pages away to its VMs on demand without any constraints. In contrast to greedy-local, greedy-remote issues a Grant-Any to the MM–M requesting 1000 of pages when the MM detects that the node needs memory.

Algorithm 2 shows the first level of *TLP*. It uses the statistics provided by the hypervisor ($memstats$, in Table I), checks the number of VMs running (line 5) and if the

Algorithm 2 First-Level of TLP

```
1: function FLM_POLICY(memstats, node_info, P, threshold)
2:     ttmem ← memstats.total_alloc_pages
3:     node_id ← node_info.id
4:     sumtgt ← 0
5:     for i ← 1, memstats.vm_count do
6:         ptr ← memstats.vm[i].puts_total_rate
7:         psr ← memstats.vm[i].puts_succ_rate
8:         puts_fail_rate ← ptr − psr
9:         if puts_fail_rate > 0 then
10:            ctgt ← memstats.vm[i].mm_target
11:            mm_target ← ctgt + (P * ttmem)/100
12:        else
13:            ctgt ← memstats.vm[i].mm_target
14:            curr_use ← memstats.vm[i].tmem_used
15:            difference ← ctgt − curr_use
16:            if difference > threshold then
17:                mm_target ← ((100 − P) * ctgt) /100
18:            else
19:                mm_target ← ctgt
20:        mm_out[i].vm_id ← memstats.vm[i].vm_id
21:        mm_out[i].mm_target ← mm_target
22:        sumtgt ← sumtgt + mm_target
23:    if sumtgt > ttmem then
24:        send_msg(Grant-Any(node_id, (sumtgt−ttmem)))
25:        for i ← 1, memstats.vm_count do
26:            nt ← (ttmem/sumtgt)*mm_out[i].mm_target
27:            mm_out[i].mm_target ← nt
28:    send_hypercall(Mem-Limit(mm_out))
29:    send_msg(Statistics(memstats) )
```

VMs have failed $puts$ since receiving the previous statistics set (lines 6–8). Initially, all VMs have an equal portion of the node's available Tmem. If $put\_fail\_rate > 0$, the MM increases the allocation of the VM by $P\%$ of the total amount of Tmem pages available to the node (lines 9–11).

If $put\_fail\_rate = 0$, the VM allocation is reduced by $P\%$ assuming that it currently has more pages allocated than its actual use by a $threshold$ ($difference > threshold$). In case not, then it keeps the current allocation (lines 12–19).

The MM then sums and assigns the target allocations to the corresponding data structures (lines 20–22). If the sum exceeds the amount of Tmem available, the MM does the following: 1) send a Grant-Any command to the MM–M to request memory pages (line 24), and 2) readjust the VMs' allocations according to Eq. 2 (lines 25–27). The MM sends the Grant-Any command because the node is under memory pressure, so it needs remote pages. There's no guarantee it will get them, thus it's necessary to readjust the allocations until more pages are available. After this, the MM issues the Mem-Limit hypercall (line 28) and the Statistics message to the MM–M if the node is a slave (line 29).

Upon receiving the Grant-Any request, the MM–M decides how many pages to actually request and from which node(s) to get them from. This is the second-level of TLP shown in Algorithm 3. The MM–M has an extended version of $memstats$, where it stores its own statistics and from other slave nodes, that it gets through Statistics messages.

Algorithm 3 Second-level of TLP

```
1: function SLM(memstats, GrAny, R, np ← {})
2:     for i ← 1, memstats.node_count do
3:         np[i].id ← memstats[i].node_id
4:         np[i].tmem ← memstats[i].total_tmem
5:     sort_nodes(np)
6:     for i ← 1, memstats.node_count ∧ GrAny.n > 0 do
7:         diff ← np[i].tmem − GrAny.n
8:         if diff > R then
9:             nr ← GrAny.n
10:            send_msg(Grant-Fwd(nr, np[i].id, GrAny.x))
11:            GrAny.n ← 0
12:        else if diff < R ∧ np[i].tmem ≥ R then
13:            nr ← np[i].tmem − R
14:            send_msg(Grant-Fwd(nr, np[i].id, GrAny.x))
15:            GrAny.n ← GrAny.n − nr
```

Algorithm 3 uses a data structure (np) that stores the amount of Tmem pages available in each node and their IDs (lines 3–4). Then, it sorts the nodes (using merge sort) in descending order in np (line 5). This is to decide which node to forward the request to (lines 6–15), usually the one with more memory available. The MM–M first checks that the potential donor node will have enough memory for itself (above a threshold R) . If the node is able to satisfy the request while meeting the threshold, then the request is forwarded to it using a Grant-Fwd message (lines 7–11).

If the node is unable to satisfy the request but has memory available, the MM–M can still forward the request (line 14) but requesting less pages (line 13) to meet the threshold of the node. Then, a new node from np is selected to request the rest of the remaining pages. From this process, it's clear that there's no guarantee that requests for pages will be satisfied.

E. Hardware Support for Memory Aggregation

vMCA requires hardware with the following features:

1) Fast interconnect providing a synchronous interface to the NUMA distributed memory.
2) Direct memory access from the hypervisor to all the memory available: the hypervisor has to be able to access transparently its local and remote memory through the same mechanisms, either through load/store instructions and/or through RDMA.
3) Remote access to a node's pages is disabled on hardware boot. Access is enabled when the node joins vMCA, when it sends the Register command.
4) Extraction of the node ID given a physical address, for instance depending on some higher-order bits.

IV. Experimental Methodology

We evaluated vMCA in a platform consisting of four computing nodes. UNIMEM-based architectures usually have 4 to 8 nodes sharing the global physical address space. Every node and VM runs Ubuntu 14.04 with Linux kernel 3.19.0, and Xen 4.5. The MM of the nodes communicate using TCP/IP sockets over Ethernet. Node 2 executes the MM–M. The hardware properties of the nodes are given in Table III.

Node	CPU	Frequency	Memory
Node 1	AMD FX Quad-Core	1.4 GHz	8 GB
Node 2	Intel Core i7	2.10 GHz	16 GB
Node 3	Intel Xeon	2.262 GHz	64 GB
Node 4	AMD FX-4300 Quad-Core	3.8 GHz	8 GB

Table III. HARDWARE CHARACTERISTICS

Accesses to remote sections of the shared global address space are emulated using the node's local memory. Xen has been modified to start using a portion of the physical memory, equal to the emulated capacity of the node. The rest of the node's memory capacity was reserved to emulate remote accesses. Whenever the hypervisor performs an "emulated" remote access, we add a delay in the hypervisor lasting 50 µs to model a reasonable worst-case hardware latency. We evaluate vMCA using the CloudSuite 3.0 Benchmarks [11], with every VM having 1 vCPU and every node having 1 GB of Tmem initially available. We run at most three VMs simultaneously with memory intensive applications that generate memory pressure on the node, and refer to each set of VMs as a *scenario*, shown in Table IV. We run each scenario at least four times.

V. RESULTS

A. Results for Scenario 1

Figures 3(a, b) show the average running times of each VM for Scenario 1 for all policies, showing average improvements of 11.94% and 12.9% in nodes 3 and 4, respectively, when going from greedy-local to greedy-remote. TLP shows an improvement of 7.3% ($P = 2.0\%$) and 7.0% ($P = 6.0\%$) in nodes 3 and 4 over greedy-remote.

Figures 3(c, d, e) present the Tmem capacity (remote and local) that each VM takes for all policies in node 3. For greedy-local and greedy-remote (Figures 3(c, d)), VM3 and VM4 take a smaller proportion of Tmem when compared to VM2, but this disparity reduces with TLP, as shown in Figure 3(e). TLP tries to be *fair* on the distribution of the Tmem capacity among the VMs.

For some values of P, TLP degrades the performance of in-memory analytics. Consider the first iteration of VM1 in node 3 for $P = 1\%$ (tlp-1, Figure 3(a)). With TLP, limits are enforced on the Tmem capacity of a VM. When a VM tries to go beyond its initial allocation, the hypervisor prevents it from taking pages until the MM updates its allocation target. But with $P = 1\%$, the targets increase slower than needed, forcing the VM to access the disk device.

With higher P, the targets increase faster. But when P becomes too high, a VM might have excess memory allocated, making it unlikely for other VMs to take a fair share, thus degrading performance. In Figures 3(a), the performance improves when increasing P up to 2.0% but degrades again as P keeps increasing. This highlights a trade-off between the adaptability of TLP to adjust to changes in memory demand and to achieve a fair distribution of Tmem.

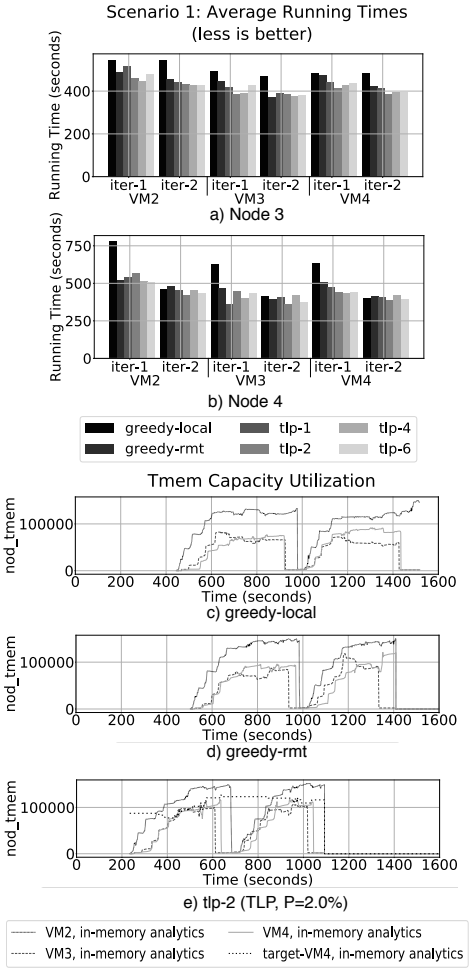

Figure 3. Running time for Scenario 1 in nodes 3 (a) and 4 (b). Tmem capacity (nod-tmem) used by every VM in node 3 for Scenario 1 in all three policies (c, d ,e). *target-VM4* refers to the target allocation of VM4.

B. Results for Scenario 2

Figures 4(a, b) present the average running times of each VM for Scenario 2 for the three policies. They show average improvements of 50.84% and 45.96% in nodes 3 and 4, respectively, when going from greedy-local to greedy-remote. With TLP, there is a further improvement of 32.1% ($P = 2.0\%$) and 31.8% ($P = 2.0\%$) in nodes 3 and 4, respectively, over greedy-remote.

Figures 4(c, d, e) show the amount of Tmem taken by each VM on node 3, for each policy. When $P = 2.0\%$, the MM enforces *fairness* in the Tmem allocation of the VMs when comparing greedy-local to greedy-remote. The difference in Tmem ownership between VM4 and other VMs is significant for greedy-local but reduces for greedy-remote, because of the remote memory availability. In both cases, the VMs are competing for the Tmem capacity. Figures 4(a, b) highlight the *adaptability-vs-fairness* trade-off. As P increases, the performance improvements hit a minimum and then increase. For this scenario, this trend is maintained for large P, making $P = 2.0\%$ an optimal value.

Scenario	VM RAM (MB)	Description
Scenario 1	VM1: 768, VM2: 1024, VM3: 1024	Every VM executes simultaneously in-memory analytics once, sleeps 5 seconds and then executes it again. The data set was taken from [20]
Scenario 2	VM1: 768, VM2: 768, VM3: 768	Every VM executes simultaneously graph analytics once. The data set was taken from [17–19], chosen to generate enoug memory pressure
Scenario 3	VM1: 768, VM2: 768, VM3: 1024	VM1 and VM2 execute graph analytics while VM3 executes in-memory analytics, all at once. Every VM executes its benchmarks twice.
Scenario 4	VM1: 768, VM2: 512, VM3: 512	VM1 executes graphs analytics, while VM2 and VM3 execute the client and server (memcached [16]) for data-caching, respectively.

Table IV. LIST OF SCENARIOS USED FOR BENCHMARKING.

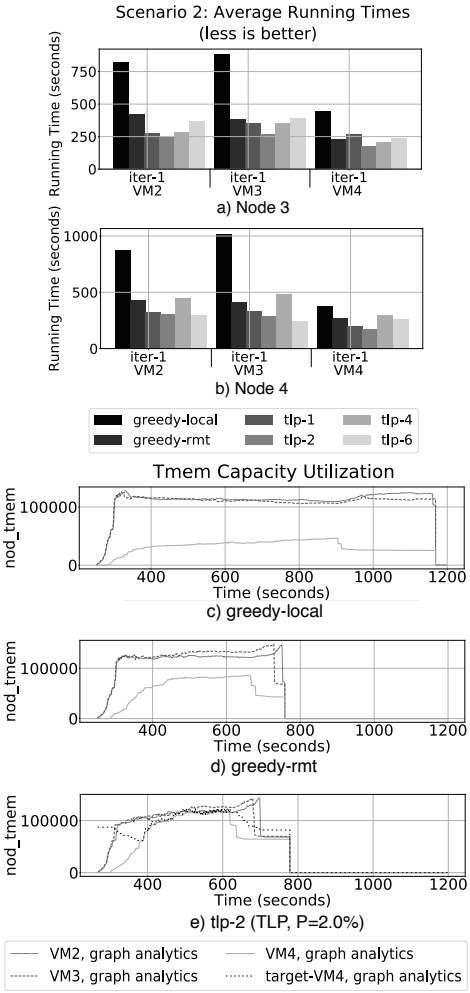

Figure 4. Running time for Scenario 2 in nodes 3 (a) and 4 (b). Tmem capacity (nod-tmem) used by every VM in node 3 for Scenario 2 in all three policies (c, d ,e). *target-VM4* refers to the target allocation of VM4.

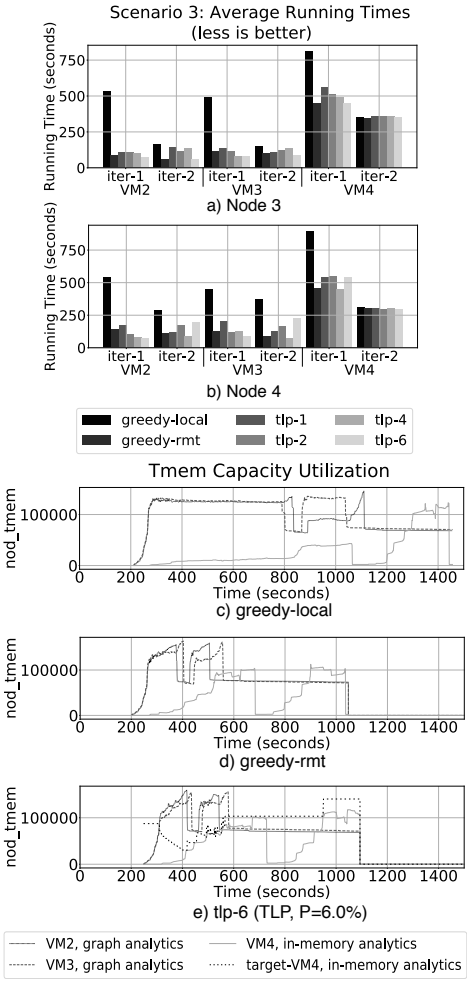

Figure 5. Running time for Scenario 3 in nodes 3 (a) and 4 (b). Tmem capacity (nod-tmem) used by every VM in node 3 for Scenario 3 in all three policies (c, d ,e). *target-VM4* refers to the target allocation of VM4.

C. Results for Scenario 3

Figures 5(a, b) show the average running times of each VM for Scenario 3 for the three policies. They show average improvements of 50.6% and 55.4% in nodes 3 and 4, respectively, when going from greedy-local to greedy-remote. When implementing TLP, there is a further improvement of 10.1% ($P = 6.0\%$) and 15.6% ($P = 4.0\%$) in nodes 3 and 4 over greedy-remote.

Figures 5(c, d, e) show the Tmem capacity taken by each VM for all policies in node 3. As remote memory becomes available to the node, the running time dramatically drops while VM4, running in-memory analytics, takes the longest. The adaptability-vs-fairness tradeoff is observed in Figures 5(a, b). Performance improves as P increases, but it can change suddenly with P, as in the second iteration of VM1 and VM2 for node 3 (Figure 5(a)). In both nodes, the second iteration of VM4 (in-memory analytics) performs the same for all policies because it runs on its own, while the

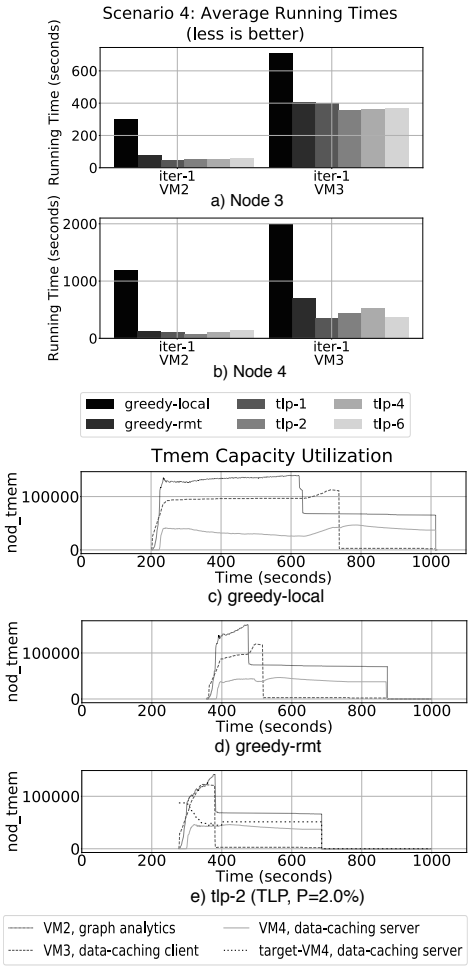

Figure 6. Running time for Scenario 4 in nodes 3 (a) and 4 (b). Tmem capacity (nod-tmem) used by every VM in node 3 for Scenario 4 in all three policies (c, d ,e). *target-VM4* refers to the target allocation of VM4.

others have completed (Figure 5(c, d, e)). In this Scenario, it is more difficult to obtain an optimal value of P.

After VM2 and VM3 execute graph analytics, they keep a large part of the Tmem pages they acquired. The MM is unable to allocate Tmem pages that are *in use*, until the VMs flushes them. But these VMs never actually flush. If the node runs more VMs with benchmarks that keep pages in this way, the Tmem pages may be inefficiently depleted.

D. Results for Scenario 4

Figures 6(a, b) show the average running times of each VM of Scenario 4 for all policies, showing average improvements of 63.5% and 76.8% in nodes 3 and 4, respectively, when going from greedy-local to greedy-remote. TLP shows improvements of 23.7% ($P = 2.0\%$) and 37.5% ($P = 2.0\%$) in nodes 3 and 4 over greedy-remote.

Figures 6(c, d, e) show the Tmem capacity of each VM for all policies in node 3. In all cases, VM4 (memcached server) uses the same amount of Tmem, while VM2 (graph analytics) and VM3 (data-caching client) compete for the Tmem capacity. With TLP (Figure 6(e)), fairness is achieved quickly, showing the adaptability-vs-fairness tradeoff in Figures 6(a, b). In this case, $P = 2.0\%$ is optimal for most VMs, except for VM3 in node 4, in which $P = 1.0\%$ is better.

VI. Related Work

Venkatesan et al. [21] use Tmem and non-volatile memory (NVM) to reduce accesses to disk by VMs in a system with DRAM and NVM, showing that Tmem is useful in systems with non-uniform memory hierarchies. The non-uniformity in our case is a consequence of the different memory hierarchies across the nodes, the amount of memory available to a node and the physical location of this memory.

Lorrillere et al. [22] proposed a mechanism, called PUMA, to improve memory utilization based on remote caching to pool VMs memory across a data center for the benefit of VMs having I/O intensive workloads. PUMA uses an interface similar to Tmem to access the cache in a remote server. vMCA differs from [22] because the latter targets clean file-backed pages for I/O intensive applications, while vMCA targets pages generated by the processes of the applications during their computations (not file-backed).

Zcache [13] is a backend for frontswap and cleancache that provides a compressed cache for swap and clean filesystem pages. *RAMster* [13] is an extension of zcache that uses kernel sockets to store pages in the RAM of remote nodes. Similarly, RAMCloud [14] is a POSIX-like filesystem in which all data is stored in RAM across the nodes. vMCA differs from all these approaches in: 1) vMCA grants and releases memory capacity at a higher granularity larger than a single page (reducing communication between nodes), 2) vMCA is not entirely within the kernel, leveraging user-space flexibility, and 3) vMCA exploits a global shared address space with low-latency communication.

Hwang et al. [23] proposed Mortar: a mechanism that pools spare memory on a server and exposes it as a volatile data cache managed by the hypervisor using a similar interface to Tmem, and it can evict objects from the cache in order to reclaim memory for other VMs. They test two use cases, the first using memcached protocol to aggregate free memory across data center, and the second works at the OS-level to cache and prefetch disk blocks. vMCA puts all the memory aggregation and management details on a user-space process, keeping the hypervisor small and secure.

VII. Conclusions and Future Work

This paper introduces vMCA, a resilient mechanism that exploits Tmem to aggregate memory capacity across multiple nodes. We evaluated vMCA using CloudSuite, obtaining up to 37.5% performance improvement when enabling memory aggregation and management policies. The results demonstrate the effectiveness of vMCA for improving the performance of the evaluated Cloudsuite applications.

Future work will investigate how to integrate in vMCA other resource management mechanisms and to create a

more complete solution for data centers. We need to consider reclamation of in-use Tmem pages, VM migration and more adaptive and predictive memory management policies.

ACKNOWLEDGEMENTS

This research has received funding from the European Union's Seventh Framework Programme for research, technological development and demonstration under grant agreement number 610456 (Euroserver). The research was also supported by the Ministry of Economy and Competitiveness of Spain (TIN2012-34557 and TIN2015-65316), HiPEAC Network of Excellence (ICT-287759 and ICT-687698), the FI-DGR Grant Program (2016FI-B-00947) of the Government of Catalonia and the Severo Ochoa Program (SEV-2011-00067) of the Spanish Government.

REFERENCES

[1] Magenheimer, D., Mason, C., McCracken, D., Hackel, K.: Transcendent memory and Linux. In: Proc. of the Linux Symposium. 2009.

[2] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A.: Xen and the art of virtualization. In: ACM SIGOPS Operating Systems Review, vol. 37. 2003.

[3] Kivity, A., Kamay, Y., Laor, D., Lublin, U., Liguori, A.: KVM: the Linux virtual machine monitor. In: Proc. of the Linux symposium. 2007

[4] VMware Virtualization for Desktop & Server, Application, Public & Hybrid Clouds http://www.vmware.com

[5] Dong, J., Hou, R., Huang, M., Jiang, T., Zhao, B., Mckee, S., Wang, H., Cui, X., Zhang, L.: Venice: Exploring Server Architectures for Effective Resource Sharing. In: Proc. of IEEE HPCA. 2016.

[6] Howard, J., Dighe, S., Hoskote, Y., Vangal, S., Finan, D., Ruhl, G., Jenkins, D., Wilson, H., Borkar, N., Schrom, G.: A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS. In IEEE ISSCC. 2010.

[7] Durand, Y., Carpenter, P., Adami, S., Bilas, A., Dutoit, D., Farcy, A., Gaydadjiev, G., Goodacre, J., Katevenis, M., Marazakis, M., Matus, E., Mavroidis, I., Thomson, J.: Euroserver: Energy Efficient Node For European Micro-servers. In: Proc. of 17th IEEE Euromicro Conference on DSD. 2014.

[8] Katrinis, K., Syrivelis, D., Pnevmatikatos, D., Zervas, G., Theodoropoulos, D., Koutsopoulos, I., Hasharoni, K., Raho, D., Pinto, C., Espina, F., Lopez-Buedo, S., Chen, Q., Nemirovsky, M., Roca, D., Klosx, H., Berends, T.: Rack-scale Disaggregated cloud data centers: The dReDBox project vision. In: Proc. of IEEE DATE. 2016.

[9] Garrido, L. A., Carpenter, P.: Aggregating and Managing Memory Across Computing Nodes in Cloud Environments. In: Proc. of 12th Workshop on VHPC. 2017.

[10] Thacker, C.: Beehive: A many-core computer for FPGAs. In: MSR Silicon Valley. 2010.

[11] Ferdman, M., Adileh, A., Kocberber, O., Volos, S., Alisafaee, M., Jevdjic, D., Kaynak, C., Popescu, A. D., Ailamaki, A., Falsafi, B.: Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In: Proc. of the 17th ACM ASPLOS. 2012.

[12] Rigo, A., Pouget, K., Raho, D., Dutoit, D., Martinez, P.Y., Doran, C., Benini, L., Mavroidis, I., Marazakis, M., Bartsch, V., Lonsdale, G., Pop, A., Goodacre, J., Colliot, A., Carpenter, P., Radojkovic, P., Pleiter, D., Drouin, D., de Dinechin, B. Paving the way towards a highly energy-efficient and highly integrated compute node for the Exascale revolution: the ExaNoDe approach. Accepted to Euromicro DSD 2017.

[13] Magenheimer. D.: Zcache and RAMster (oh, and frontswap too) overview and some benchmarking. https://oss.oracle.com/projects/tmem/dist/documentation/presentations/LSFMM12-zcache-final.pdf. 2012

[14] Ousterhout, J., Agrawal, P., Erickson, D., Kozyrakis, C., Leverich, K., Mazières, D., Mitra, S., Narayanan, A., Parulkar, G., Rosenblum, M., Rumble, S., Stratmann, E., Stutsman, R.: The case for RAMClouds: scalable high-performance storage entirely in DRAM. In: ACM SIGOPS Operating Systems Review, vol. 43. 2010.

[15] Svärd, P., Hudzia, B., Tordsson, J., Elmroth, E.: Hecatonchire: Towards Multi-host Virtual Machines by Server Disaggregation. In: Euro-Par International Parallel Processing Workshops. 2014.

[16] memcached - a distributed memory object caching system. http://memcached.org

[17] Ross, R.A., Ahmed, N.K.: The Network Data Repository with Interactive Graph Analytics and Visualization. In: Proc. of the 29th AAAI. 2015.

[18] Ross, R.A., Ahmed, N.K.: An Interactive Data Repository with Visual Analytics. In: SIGKDD Explor., vol 17. 2016.

[19] Cho, E., Myers, S.A., Leskovec, J.: Friendship and mobility: user movement in location-based social networks. In: Proc. of the 17th ACM SIGKDD. 2011.

[20] Harper, F. M., Konstan, J.A.: The MovieLens Datasets: History and Context. In: ACM TIIS. 2015.

[21] Venkatesan, V., Qinqsong, W., Tay, Y.C.: Ex-Tmem: Extending Transcendent Memory with Non-volatile Memory for Virtual Machines. In Proc. of the IEEE HPCC, IEEE 6th ICSS, IEEE 11th ICESS. 2014.

[22] Lorrillere, M., Sopena, J., Monnet, S., Sens, P.: Puma: Pooling Unused Memory in Virtual Machines for I/O Intensive Applications. In Proc. of the 8th ACM SYSTOR. 2015.

[23] Hwang, J., Uppal, A., Wood, T., Huang, H.: Mortar: Filling the Gaps in Data Center Memory. In Proc. of the 10th ACM SIGPLAN/SIGOPS VEE. 2014.