Aggregating and Managing Memory Across Computing Nodes in Cloud Environments

Luis A. Garrido and Paul Carpenter

Barcelona Supercomputing Center,
C/ Jordi Girona. 31, 08034 Barcelona, Spain
{luis.garrido,paul.carpenter}@bsc.es
http://www.bsc.es

Abstract. Managing memory capacity in cloud environments is a challenging problem, mainly due to the variability in virtual machine (VM) memory demand that sometimes can't be met by the memory of one node. New architectures have introduced hardware support for a shared global address space that, together with fast interconnects, enables resource sharing among multiple nodes. Thus, more memory is globally available to a computing node avoiding the costly swaps or migrations. This paper presents a solution to aggregate the memory capacity of multiple nodes in a virtualized cloud computing infrastructure. It is based on the Transcendent Memory (Tmem) abstraction and uses a user-space process to manage the memory available to a node, and distribute the aggregated memory across the computing infrastructure. We evaluate our solution using CloudSuite 3.0 benchmarks on Linux and Xen.

Keywords: Virtualization; Simulation, Modeling and Visualization

1 Introduction

Current data centres use large numbers of servers provisioned with their own computing resources. These servers share none of their resources, and they communicate over an Ethernet (or similar) network. Newer system architectures try to improve over this resource isolation approach, allowing servers to share their resources through the memory hierarchy. These systems provide a shared global physical address space, accessing memory at low latency using very fast interconnects. These systems are composed of *coherence islands* (*nodes*), with cache coherency enforced within an island, with no global hardware coherence.

Each node's physical memory capacity is distributed by the hypervisor among one or more Virtual Machines (VMs). The demand for memory resources generated by the VMs vary due to the different workloads they execute over time. To improve utilization of the memory capacity of the node, physical memory is often overcommitted, which causes a VM to have less memory than the amount it was configured at boot time. The physical memory given to a VM is usually adjusted using memory ballooning and/or memory hotplug. Xen's Transcendent

Memory (Tmem) [1] is another way to make memory capacity available to the VMs, through a paravirtualized put–get interface.

This paper presents a mechanism, called GV-Tmem (Globally Visible Tmem), that extends the hypervisor to share the memory capacity of the nodes across the computing infrastructure. GV-Tmem introduces minimal changes to the hypervisor, keeping it small, secure and self-contained. Most of the complexity is in a user-space memory manager process running in the privileged domain that supports memory management policies, inter-node communication and dynamic addition and isolation of nodes. Our main contributions are:

1. A software architecture to aggregate memory across nodes using Tmem.
2. A two-tier mechanism for allocation and management of aggregated memory.

This paper is organized as follows. Section 2 gives the necessary background on virtualization, Tmem and coherence islands. Section 3 explains GV-Tmem. Section 4 describes the experimental methodology and Section 5 shows our results. Section 6 compares with related work and Section 7 concludes the paper and outlines future work.

2 Background

2.1 Virtualization In IaaS Clouds

Cloud computing provides on-demand access to an apparently unlimited pool of computing resources. There are multiple cloud computing service models, but the most fundamental is Infrastructure-as-a-Service (IaaS). IaaS shares the underlying hardware memory resources among customers using virtualization software known as a hypervisor.

Virtualization and Memory Management The hypervisor virtualizes the physical resources, including memory, of the node. It creates and manages Virtual Machines (VMs), each of which runs its own (guest) Operating System (OS). When a VM is created, the hypervisor allocates to it a portion of the physical memory capacity. If a VM later requires more memory (*memory underprovisioning*) it will generate accesses to its (virtual) disk device, even if some of the node's memory is unassigned or sits idle in a VM that does not need it. When a VM has more memory than needed (*memory overprovisioning*), then the memory is underutilized and will be used for disk caches. In both cases, it is beneficial to re-allocate memory, making it available to the VM that needs it.

There are solutions for dynamically re-allocating memory among VMs, including memory ballooning and memory hotplug, both implemented in the Xen and Linux. These mechanisms have been widely deployed in current data centres, with significant performance benefits in terms of higher memory utilization. However, they do not provide adequate interfaces to aggregate memory capacity across multiple nodes.

State-of-the-Art Transcendent Memory Transcendent memory (Tmem) [1] is another memory management mechanism that pools the idle or unassigned physical pages. Tmem is abstracted as a key–value store in which pages are accessed through a put–get interface, a put operation to write a page in the store that becomes mapped to the VM that issued it, and a get operation to read pages back. The Tmem interface supports also flush-page and flush-object operations, which return pages to the tmem pool. In order to enable Tmem, the VMs need to have a Tmem kernel module (TKM) which handles all accesses to the Tmem pages on behalf of the VM by issuing hypercalls to the hypervisor.

2.2 Hardware Support for Coherence Islands

In systems such as Venice [2] and EUROSERVER [3] (based on ARM), the processors in each node are connected in clusters via a local cache-coherent interconnect to local resources. Remote memory is visible through the global physical address space, and communication across nodes is achieved through an *inter-node interface* and *global interconnect*. Other examples of similar architectures include dRedBox [4] and Beehive [5]. The essential characteristics of these architectures are:

- Each node executes its own hypervisor and OSs.
- Global physical memory address space with low-latency access.
- Routing is based on the global physical address (e.g. high-order bits).
- Fast communication is provided across the nodes of the system.

3 GV-Tmem design

GV-Tmem consists of three software components:

- Extended Xen Hypervisor (Section 3.1)
- Tmem Kernel Module in the kernel of all domains (Section 3.2)
- Memory Manager (MM) in user space in Dom0 (Section 3.3)

3.1 Xen Hypervisor with Extensions

The hypervisor extensions for GV-Tmem are minimum and localized in the Tmem subsystem. First, the hypervisor enforces the memory allocation constraints determined by the MM. Second, it allocates and deallocates physical pages and passes ownership of blocks of pages in and out of the hypervisor. Third, it collects information of tmem utilization that sends to the MM.

Enforcing local per-VM memory constraints: The hypervisor constrains the Tmem consumption of its VMs, based on the allocation determined by the MM. The MM specifies the maximum number of pages a VM can use.

Page allocation and transfer of ownership: GV-Tmem ensures that each physical page is *owned* by at most one hypervisor. Tmem pages owned by a hypervisor are allocated using a zoned Buddy allocator, with a zone for each node from which it has ownership of at least one page.

A Tmem put operation allocates the closest free page from the allocators. A Tmem flush operation causes a page to be returned to the corresponding Buddy allocator. A Grant hypercall is used when the MM receives ownership of a list of blocks (each an appropriately-aligned power-of-two number of pages). These are added as free blocks to the appropriate Buddy allocator. In contrast, a Request hypercall is used to release ownership of pages on behalf of another node.

Memory statistics: The hypervisor collects information about the tmem utilization of the VMs, such as number of put, get and flush operations, and how many of them fail or succeed. The information gathered needs to be minimum to avoid communication overhead from the hypervisor to the MM.

3.2 Tmem Kernel Module (TKM)

Interfacing to the Tmem client interface requires a kernel module in each VM, but the kernel module in the privileged domain (Dom0) acts only as an interface between the hypervisor and the node's MM (using netlink sockets).

3.3 Dom0 User-space Memory Manager (MM)

Each node has a user-space Memory Manager (MM) in Dom0. The MMs perform most of the work of GV-Tmem by cooperating to:

1. Distribute memory owned by each node among its guests
2. Distribute global memory capacity among nodes
3. Implement the flow of page ownership among nodes
4. Enable nodes to join and leave, and handle failures

Joining the GV-Tmem system There is one MM Master (MM–M) that controls the system and distributes the global memory capacity. The messages passed among the MMs are listed in Table 1. A node requires a configuration file, which provides the network addresses of all nodes, their mappings to a node ID and credentials to establish secure connections. When a node R wishes to join the GV-Tmem system, it sends Register to the MM–M (see Table 1). The MM–M sets R's state to Active. Then it sends a Enable-Node(R,1) message to all registered nodes. Every node maintains a bitmap of the active nodes.

Distributing memory owned by a node among guests: The local MM determines the maximum number of pages for each VM. This is done using a policy that determines this maximum based on the statistics sent from the hypervisor. This is the first tier of the memory management strategy. Pages are distributed subject to a memory consumption limit, set by the MM–M using the Mem-Limit message.

Distributing global memory capacity among nodes: Nodes in the Active state, regularly send Statistics messages to the MM–M. Based on these and the global memory policy, the MM–M redistributes the memory among nodes by sending the Grant-Any message to the donor node, which is a request to transfer ownership of a number of free physical pages to a given node. This is the second tier of the memory management strategy.

Command	Direction	Description	Slave state
Distribution of global memory capacity			
Statistics(S)	SL→MT	Send node statistics S to Master	A
Grant-Any(n, x)	MT→SL	Request grant of n pages to slave x	A
Grant-Return(n, x)	MT→SL	Request grant of n pages to slave x	A
Force-Return(x)	MT→SL	Disable node x and return pages located at x	A
Mem-Limit(n)	MT→SL	Limit allocated pages to store local data	A
Flow of page ownership			
Grant($b, \cdots$)	SL→SL	Transfer ownership of blocks of pages	A
Node state changes			
Register	SL→MT	Register a new node	$I \to A$
Leave-Req	SL→MT	Node requests to leave or shutdown	$A \to L$
Leave-Notify	MT→SL	MM–M notifies that the recipient has left	$L \to I$
Enable-Node(x, e)	MT→SL	Accept ($e = 1$) or reject ($e = 0$) pages at x	A

Table 1. MM message types. *SL*: *slave*, *MT*: *master*, *I*: *Inactive*, *A*: *Active*, *R*: *Recovery*, *L*: *Leaving*

Implementing flow of page ownership: The MM–M rebalances memory capacity, without knowing the physical addresses. Ownership of physical addresses is transferred in a peer-to-peer way using Grant messages.

Leaving the GV-Tmem system To cleanly shutdown a node R that is in GV-Tmem, the following procedure must be followed.

1. Node R sends a Leave-Req message to the MM–M.
2. Upon receiving the Leave-Req, the MM–M sets the node to Leaving state and sends Force-Return(R) to all nodes. The nodes return the pages at R that they own and will reject any pages received in future Grant messages.
3. Node R frees all pages used by Tmem and returns ownership of all remote pages to their home nodes.
4. Periodically, each node sends Grant messages to node R to return ownership of the pages that it had borrowed.
5. Once the MM–M has received Statistics messages from all nodes indicating that R is disabled and that it owns no pages at R, the MM–M moves R to Inactive and sends Leave-Notify to R.
6. At this point, the node R may shutdown.

Hardware Support for Memory Aggregation For proper functioning, GV-Tmem requires the underlying hardware to provide the following features:

1. A fast interconnect, providing a synchronous interface across the system.
2. Direct memory access from the hypervisor to all the memory available.
3. Remote access to a node's pages is disabled on hardware boot.
4. Given a physical address, it must be possible to extract the Node ID.

Node	CPU	Frequency	Memory
Node 1	AMD FX Quad-Core	1.4 GHz	6 GB
Node 2	Intel Core i7	2.10 GHz	8 GB
Node 3	Intel Xeon	2.262 GHz	64 GB

Table 2. Hardware characteristics

Scns.	VM Parameters	Description
Scn. 1	VM1, VM2: 768 MB RAM, 1 CPU; VM3: 1 GB RAM, 1 CPU	All VMs execute in-memory-analytics, sleep 5 seconds and then execute it again.The data set used is from [7].
Scn. 2	VM1, VM2, VM3: 512 MB RAM, 1 CPU	VM1 and VM2 execute *usemem*, and VM3 starts when VM1 allocates 640 MB.
Scn. 3	VM1, VM2: 512 MB RAM, 1 CPU	Every VM executes graph-analytics once. They use the dataset provided by [8], [9], [10].

Table 3. List of scenarios used for benchmarking

4 Experimental Methodology

We tested GV-Tmem in a platform consisting of three nodes. Every VM runs Ubuntu 14.04 with Linux kernel 3.19.0+ as the OS, and Xen 4.5. The nodes communicate using Ethernet TCP/IP sockets. Node 2 acts as the Master node and executes no VMs. Table 2 summarizes the hardware properties of the nodes.

The shared global address space was emulated using the node's local memory. We modified Xen to start up using a portion of the physical memory capacity, equalling the emulated memory capacity of the node. The rest of the node's memory capacity was reserved to emulate remote data storage. Whenever the hypervisor performs an "emulated" remote access, we add a delay in the hypervisor lasting 50 µs to model hardware latency.

We evaluate GV-Tmem using CloudSuite 3.0 [6]. We also designed a microbenchmark called *usemem*, which allocates a varying amount of memory, starting with 128 MB. Every time it allocates memory, it performs a series of write/read operations while checking the correctness of the values read in the allocated memory. We execute at most three DomUs simultaneously, and refer to each set of DomUs as a *scenario* (or *Scn*). Table 3 shows the scenarios used. For Scns. 1 and 3, all nodes have 1 GB of Tmem capacity. For Scn. 2, Node 2 has 1 GB of Tmem, while Nodes 1 and 3 have 384 MB.

This paper uses three memory management policies:

- **greedy-local**: Default policy used in Tmem with only local memory, which gives memory away on demand. No maximum values are set to limit the amount of memory taken by a VM.
- **greedy-remote**: An extended version of greedy-local using remote memory.
- **TTM**: A two-tier memory management strategy that allocates memory locally for each VM (first-tier) depending on the node's statistics, and issues

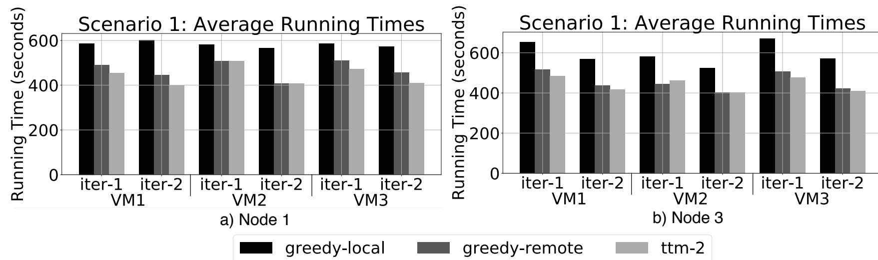

Fig. 1. Running time for Scn. 1 in nodes 1 and 3. Time is in seconds (less is better).

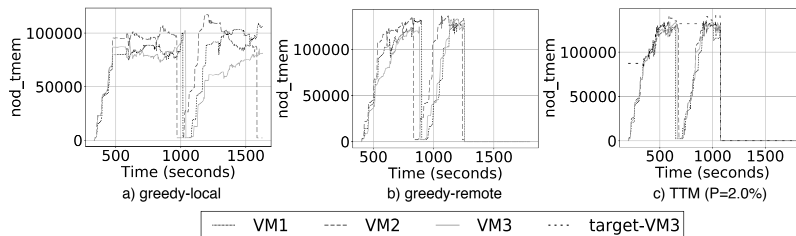

Fig. 2. Tmem capacity (nod-tmem) obtained by every VM in node 3 for Scn. 1. The label *target-VM3* refers to the target allocation of VM3.

requests for remote memory (second-tier) depending on the perceived memory pressure. The pages allocated and deallocated to a VM are increased by a percentage $\%P$ of the pages owned by the node (local or remote).

5 Results

Results for Scenario 1 Figure 1 shows the average running times of each VM for Scn. 1. The running time improves by an average of 19.4% and 23.5% in nodes 1 and 3, respectively, when going from greedy-local to greedy-remote. When implementing TTM with $P = 2.0\%$, there is further improvement of 6.0% and 4.0% over greedy-remote, demonstrating the need to implement memory management policies when there is significant memory pressure.

Figure 2 shows the amount of Tmem capacity that each VM is able to use for the three policies mentioned in node 3. With greedy-local (Figure 2(a)), VM3 in both iterations cannot obtain a fair share of the available Tmem capacity. With greedy-remote (Figure 2(b)), the VMs are able to get more total Tmem, but because of the lack of memory management policies, some VMs are unable to obtain a fair share of Tmem. With TTM (Figure 2(c)), every VM is ensured a fair amount of the available Tmem, demonstrating that TTM is able to ensure fairness regarding the VMs' allocation of tmem, improving the running times.

Results for Scenario 2: the Usemem Scenario The average running times for Scn. 2 are shown in Figure 3. When enabling greedy-remote, VM3,

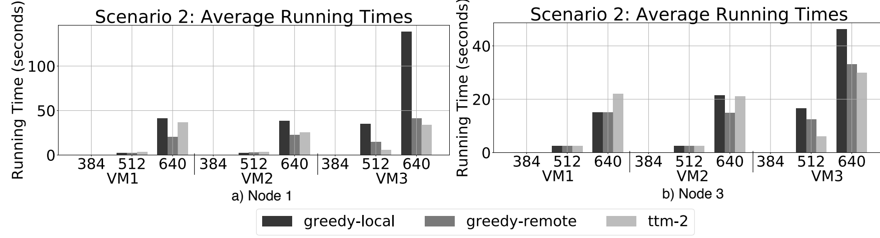

Fig. 3. Running time for Scn. 2 for nodes 1 and 3.

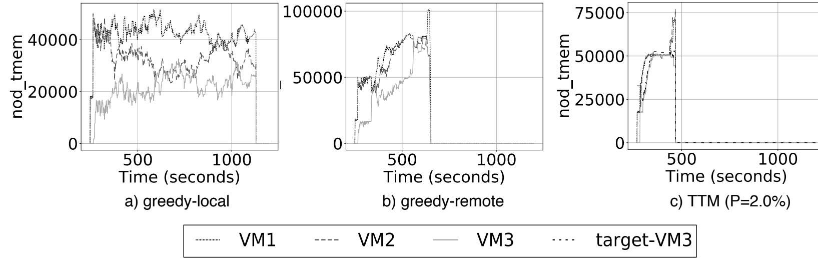

Fig. 4. Tmem capacity (nod-tmem) obtained by every VM in node 3 for Scn. 2.

VM2 and VM1 reach an average performance improvement of 63%, 20% and 13% respectively. When TTM is enabled, VM3 shows a maximum and minimum improvement of 27% and 5.5% in node 1, respectively, and a maximum and minimum improvement of 51% and 9.3%, respectively, in node 3. However, VM1 and VM2 both experience a performance reduction.

Figure 4 shows the remote memory capacity that each VM is using for the three policies. Figure 4(a) shows that VM3 struggles to obtain memory pages using *greedy-remote*. This is similar to the case in Figure 2(a), in which the VM3 was unable to reach its fair share of Tmem. With TTM, VM3 obtains a larger amount of Tmem, improving its performace. Here, VM3's improvement comes at the expense of VM1 and VM2, balancing the Tmem pages every VM can have.

Results for Scenario 3 The average running times for Scn. 3 are shown in Figure 5. In this case, node 1 improves by a maximum and a minimum of 92.3% and 92.1%, respectively, when comparing greedy-local to greedy-remote. When enabling TTM, it improves by a maximum and a minimum of 6.0% and 0.9%.

In node 3, there's a maximum and mininum improvement of 84.4% and 83.1% when comparing greedy-local to greedy-remote. However, performance degrades by 10% with TTM compared to greedy-remote. When enabling TTM, the VMs require more memory but TTM enforces limits, although flexible, on the memory they can take, similar to what occurs in Scn. 2 for VM1 and VM2. When disabling TTM, the VMs take memory unrestrained thus performing slightly better than TTM. This highlights the need for more adaptive memory management policies.

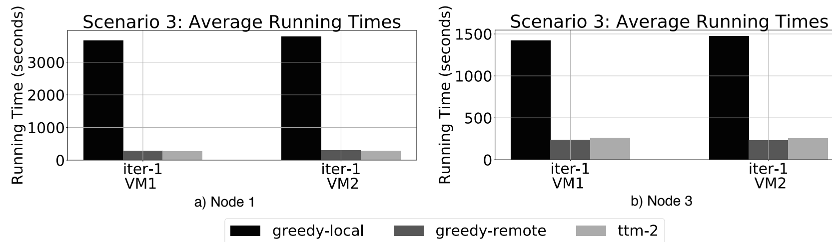

Fig. 5. Running time for Scn. 3 for nodes 1 and 3.

6 Related Work

Zcache [11] is a backend that provides a compressed cache for swap and clean filesystem pages. *RAMster* [11] is an extension of zcache that uses kernel sockets to store pages in the RAM of remote nodes. In contrast, our approach grants and releases blocks of pages at greater granularity, reducing the amount of software communication between nodes, since we exploit the shared global address space. RAMster is implemented in the kernel whereas our approach uses a user-space process in a privileged VM, providing greater flexibility for memory management. RAMCloud [12] is a POSIX-like filesystem in which all data is stored in DRAM across nodes, placing the data in one or more nodes, introducing issues of global coherency and exclusivity of access. Our solution aggregates memory exploiting a global shared address space, without requirements for global coherency. Hecatonchire [13] achieves resource aggregation by decoupling virtual resource management from physical resources. It uses a mediation layer that arbitrates how applications access resources. We differ from [13] by making memory available to the hypervisor through a user-space process.

7 Conclusions and Future Work

This paper introduces GV-Tmem, a method that exploits Tmem to share memory capacity across multiple nodes. We evaluated GV-Tmem using CloudSuite, obtaining up to 51% performance improvement using simple memory management policies. The results demonstrate the effectiveness of GV-Tmem, and the need for two-tier memory management strategies within and across nodes.

Future work will investigate how to integrate GV-Tmem with other resource management mechanisms of other cloud software. It is also necessary to develop more sophisticated two-tier global memory management policies, in order to improve adaptivity and responsiveness to changes in memory demand.

8 Acknowledgements

This research has received funding from the European Unions 7th Framework Programme (FP7/20072013) under grant agreement number 610456 (Euroserver).

The research was also supported by the Ministry of Economy and Competitiveness of Spain under the contract TIN2012-34557, HiPEAC-3 Network of Excellence (ICT- 287759), and the FI-DGR Grant Program (file number 2016FI_B 00947) of the Government of Catalonia.

References

1. Magenheimer, D., Mason, C., McCracken, D., Hackel, K.: Transcendent memory and Linux. In: Proc. of the Linux Symposium, pp. 191–200. Citeseer, 2009
2. 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: IEEE Intl. Symp. on High-Performance Computer Architecture (HPCA), 2016.
3. 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: 17th Euromicro Conference on Digital System Design (DSD), pp. 206–2013. IEEE (2014).
4. 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: Design, Automation & Test in Europe Conference & Exhibition (DATE). IEEE (2016).
5. Thacker, C.: Beehive: A many-core computer for FPGAs. In: MSR, Silicon Valley. 2010.
6. 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 Intl. Conf. on Architectural Support for Programming Languages and Operating Systems, pp. 37–48. ACM, England (2012).
7. Harper, F. M., Konstan, J.A.: The MovieLens Datasets: History and Context. In: ACM Trans. on Interactive Intelligent Systems, pp. 19:1-19:19. ACM, USA (2015).
8. Rossi, R. A., Ahmed, N. K.: soc-twitter-follows - Social Networks. http://networkrepository.com/soc-twitter-follows.php.
9. Ross, R.A., Ahmed, N.K.: The Network Data Repository with Interactive Graph Analytics and Visualization. In: Proc. of the 29th AAAI Conference on AI. 2015.
10. Ross, R.A., Ahmed, N.K.: An Interactive Data Repository with Visual Analytics. In: SIGKDD Explor., vol 17, no. 2, pp. 37–41. 2016.
11. 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
12. 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: SIGOPS Operating Systems Review, vol. 43, pp. 92–105. ACM (2010).
13. Svärd, P., Hudzia, B., Tordsson, J., Elmroth, E.: Hecatonchire: Towards Multihost Virtual Machines by Server Disaggregation. In: Euro-Par International Parallel Processing Workshops, pp. 519–529. Springer International Publishing, Portugal (2014).