Multi-site synchronous VM replication for persistent systems with asymmetric read/write latencies

Vasily A. Sartakov
TU Braunschweig
Braunschweig, Germany
Email: sartakov@ibr.cs.tu-bs.de

Rüdiger Kapitza
TU Braunschweig
Braunschweig, Germany
Email: rrkapitz@ibr.cs.tu-bs.de

***Abstract*—**

Novel non-volatile memory is considered as a future replacement for conventional main memory. While besides persistence non-volatile memory technologies promise higher storage density and lower power demand, they also possess an asymmetry between fast read and slow write access times. The latter can be in the order of *2x* to even *20x*. While persistent main memory per se asks for novel system software support, the asymmetry between read and write access times needs to be addressed for building efficient solutions.

This paper addresses virtual machine replication for high availability when non-volatile memory is put in use. So far *asynchronous VM replication* that stops a virtual machine, performs a local copy of dirty pages, resumes the virtual machine, before the data is transferred to a back-up site, is the state of the art. In contrast, we propose a *multi-site zero-copy synchronous VM replication*, which utilizes Remote Direct Memory Access and is tolerant to different read/write latencies. We demonstrate that for future hardware settings synchronous VM replication provides a performance increase of up to 27% compared to current best practices.

***Index Terms*—Persistent Systems; Replication; Virtualization; RDMA; NV-RAM;**

I. Introduction

In-memory computing builds a de-facto standard for low-latency data center applications. To address growing workload sizes, servers are equipped with more and more main memory. While this trend presumably will continue, the attached rising energy demand and scaling limitations of conventional DRAM require a search for alternatives.

As a possible escape route novel non-volatile memory technologies are considered, as they promise higher storage density and less energy demand. In fact, first initiatives in industry and research such as *The Machine*[1] from HP and *Firebox* [1], propose the use of universal byte-addressable persistent memory, i.e. memory, which provides DRAM-like performance and storage-level non-volatile capacity [2].

Systems based on non-volatile universal memory differ from current architectures that utilize primary and secondary storages in the following aspects: Firstly, main memory is now persistent and recent works focus on efficiently utilizing this property ([3], [4]). Secondly, systems based on non-volatile memory technologies differ significantly from DRAM-based system, since current persistent memory technology have asymmetric read and write latencies of 2x up to even 20x ([5], [6], [7], [8]). Thus, as for example demonstrated by this work, established approaches for efficient system software need to be revisited and revised when implementing software dedicated for universal persistent systems.

As more and more internet-based services are considered as critical, High-availability (HA) approaches such as continuous replication of virtual machines are on the rise. With the adoption of universal persistent memory, fault-tolerant virtual machine replication will become even more important as all short- and long-lived data will be stored in the same place. Also the risk of data loss increases since recent works demonstrate ([9], [5], [10]), that some Non-Volatile Random-Access Memory (NV-RAM) technologies like Phase-Change RAM (PC-RAM) have lower endurance (10^8 writings) compared to DRAM (10^{15} writings).

Asynchronous replication of virtual machines, as proposed by REMUS [11], provides a HA service for virtualization platforms. In detail, the replicated virtual machine is periodically stopped, a local copy of dirty pages is performed, and afterwards the virtual machine is resumed before the copied dirty pages are transferred to a back-up site. This way high-performance replication is provided while still being able to tolerate crashes.

The great success of asynchronous VM replication is based on characteristics of memory-to-memory operations. Symmetric and low read/write latencies allow fast copying of dirty pages to a staging area and thus, enable to move the transfer of state changes from the critical path. In the case of an asymmetric read/write latencies, as typical for recent NV-RAM technologies, where write operations take up to 20 times longer than read operations [12], additional copy operations have a strong negative impact on replication performance.

In this work, we propose a *multi-site synchronous VM replication technique* optimized for rack-scale memory-centric architectures with asymmetric read/write latencies. We use modern interconnect technology like Remote Direct Memory Access (RDMA) and utilize its low-level features to realize zero-copy migration to multiple target hosts. As a result, for

[1]http://www.labs.hpe.com/research/themachine

978-1-5090-5652-1/17/$31.00

some candidate technologies of persistent memory [7], we achieve a performance increase of up to 27%.

II. High-availability techniques for Virtual Machines

Virtual machine replication is a widely used approach to achieve HA in virtual environments. It is transparent for the guest system and does not require special hardware support.

The replication scheme involves two sides: the active side performs actual execution while the passive side periodically receives the state changes of the active side. Remus [11], Kemari [13] and qemu-mc [14] are examples for this VM state synchronization technique. These projects were developed for different hypervisors (i.e. QEMU and XEN) but follow the same general scheme of asynchronous VM replication.

The asynchronous VM replication consists of four steps[2]: After several milliseconds the Virtual Machine Monitor (VMM) stops execution of the primary VM, copies dirty pages and the context of the VM to a local staging area. Next, it resumes the VM and performs copying of the dirty page set to the remote side in parallel with VM execution. On the remote side the VMM applies the dirty pages and VM context to the backup VM.

The execution of the active side, which takes place during the process of data copying and applying, is called *speculative* [11]. The speculative execution is not protected and in case of a fault during the delivery process the HA system loses the last checkpoint. Meanwhile, the asynchronous scheme is significant in decreasing performance degradation since degradation depends only on the size of the dirty set and the local copying time, but does not depend on the network performance. Network performance determines the maximum frequency of checkpoints, but not the performance of a virtualization system. With increasing of replication frequency, the count of dirty pages decreases, and, respectively, the copying and delivery time decrease too. In sum, this approach demonstrates reasonable performance degradation (31% for 10 replication phases per second), while providing high-availability [11].

A. Persistent environment

Future rack-scale memory-centric architectures such as *The Machine* from HP and *Firebox* from UC Berkeley rely on a universal memory layer based on NV-RAM technologies but also industrial consortiums, like SNIA[3] work on the development of computation architectures featuring a universal memory layer. There are multiple possible technologies for future memory-centric architectures, for example, Spin-Torque Magnetoresistive RAM (ST-MRAM), PC-RAM and RRAM. All of them have different characteristic regarding their read and write latencies as reported by recent studies ([12], [15]).

As highlighted in Tab. I, the ratio between read and write latencies reaches up to 20 times for 3D Xpoint technology, the latter being a variation of PC-RAM[4]. Another candidate technology – ST-MRAM developed by Everspin, also demonstrates strong asymmetry in read/write operations [12]. Thus, write operations become more costly compared to read operations, and this should be taken into account for novel software designs.

TABLE I
Read/write latency of recent non-volatile memory technologies

Characteristic	SDRAM	ST-MRAM	3D Xpoint	RRAM
Supplier	Samsung	Everspin	Intel/Micron	Crossbar
Write Latency	10 ns	20 ns	1 µs	100 µs
Read Latency	10 ns	10 ns	50 ns	100 ns
Write/Read ratio	1	2	20	1000

In the case of NV-RAM-based rack-scale memory-centric computing, replication approaches should be revisited to prevent unnecessary intermediate writes. Indeed, the replication techniques based on copying of dirty pages, i.e., asynchronous VM replication as described above, will cause a significant performance degradation. Solutions featuring a zero-copy behavior, for example, synchronous VM replication as proposed by work, in turn, can provide better performance.

B. Remote Direct Memory Access

Remote Direct Memory Access (RDMA) is a modern communication technology, which allows data transfer from the server to the client in a zero-copy manner. Classical approaches based on a Network Interface Controller (NIC) and TCP/IP come attached with reasonable overhead when sending/receiving data. For example, to send data a user-space application needs to pass the data to the TCP/IP stack, the stack passes the data to the device and the device delivers the data via the network. The main idea of RDMA is to offload TCP/IP from the system to the RDMA enabled NIC (RNIC) – as result, RDMA communication works in user-space without the involvement of the kernel (Fig. 1).

RDMA-based transfer is handled by the following three components: RDMA itself, DDP (Direct Data Placement) and a transport protocol. Transport could be Infiniband (IB) or ethernet (an ethernet with the support of RDMA is called RDMA over Converged Ethernet - RoCE). DDP describes mechanisms of data transfer. For example, a DDP header for a read request should consist [16] of five tags: Data Sink Steering Tag, Data Sink Tagged Offset, RDMA Read Message Size, Data Source Steering Tag and Data Source Tagged Offset. The idea of these tags is that each memory region used for RDMA has a unique identifier (ID) within the network cluster, and both communication partners have their own IDs associated with local buffers. RDMA itself is an upper layer, which *"provides the semantics to enable Remote Direct Memory Access between peers in a way consistent with the application requirements"* [17].

[2]For simplicity, replication of primary storage and I/O are omitted.

[3]Storage Networking Industry Association (SNIA), http://www.snia.org

[4]http://www.eetimes.com/document.asp?doc_id=1328682

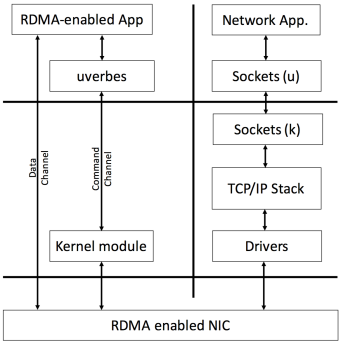

Fig. 1. Software stack for RDMA and traditional network applications

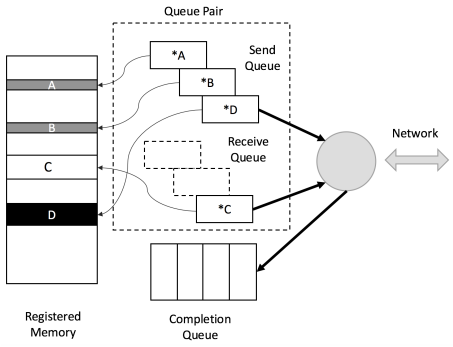

Fig. 2. Data transfer and RDMA queues

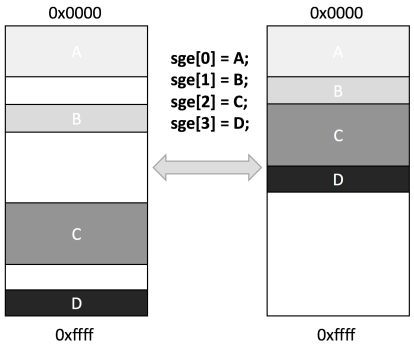

Fig. 3. Scatter/gather data transfer via RDMA

On the programming level, RDMA support is provided by a dynamically loaded library *verbs*[5], which enables mechanisms to register memory regions, an API to perform data transfer and more. The basic concept of RDMA programming is as follows: firstly, there is a preparation phase performed on both sides of a future connection. Memory regions for future transfer shall be allocated. Secondly, there is a process to establish a connection. Thereby, a connection is identified as a Queue Pair (QP). This term is conceptually close to an IP socket. In general, RDMA software deals with three queues: Sending Queue, Receiving Queue and Completion Queue (Fig. 2). The first two are the Queue Pair and are always created together. The Completion Queue is used to notify when a task in one of the work queues is completed. Creation of a QP is accompanied by the registration of memory regions. The memory regions are used for data transfer and should be registered with verbs, i.e. an RNIC should be notified that the region of virtual memory is used by an application. Then, RNIC can establish a transfer channel from hardware to the user-space program. Also, on this stage access permissions to memory regions are defined. Next, data transfer operations can be performed.

RDMA provides asynchronous communication only. Communication is performed by assigning Work Requests (WR) to internal queues, which can be checked for completion. Furthermore, asynchronous events are generated by the RNIC, which can be *affiliated*, or *unaffiliated* [6]. Affiliated events are used to notify of data delivery, while unaffiliated events are used for the detection of connection faults.

There are three general types of communication: read/write, send/receive and atomic operations. The difference between the first two types is as follows: the read/write operation requires registration of memory on both sides while the send/receive operation requires registration with the QP only on the local side. Moreover, send/receive operations generate Work Completion (WC) events on the remote side, while read/write operations generate WC events only if used in the "Immediate" (IMM) form of read/write. In other words, there are two approaches how to transfer data: two allocations on both sides or one allocation on the local side accompanied by automatic allocation on the remote side. Also, there are two methods of notification for sending: without notification (possible only with write/read) and with notification (send/receive and read/write with IMM). The third type of communication - atomic operations – is presented in two forms: *Compare and Swap* operation and *Fetch and Add* operation. We do not use this kind of operations in our work.

Another important feature of RDMA and also important for our approach is hardware support for scatter and gather (SG) data. RDMA can receive a list of memory regions from a user program, deliver all of them to the remote side and place them successively on a remote buffer without additional copying (Fig. 3). The SG data is described by a structure, which includes data offset in the virtual memory of the process, the size of the region, and the local key (identifier) of the registered buffer. The number of elements in SG list is defined by the hardware.

III. Pre-considerations regarding synchronous VM replication

Synchronous VM replication as presented in this work consists of four steps: after several milliseconds, the VMM stops the execution of the VM, identifies dirty pages, sends the pages and the context of the VM to a remote side, applies them to the backup VM, and then resumes execution. Instead of copying dirty pages to a staging area, we transfer dirty pages directly to the remote side.

A. Comparison of synchronous versus asynchronous replication

Conceptually, both synchronous and asynchronous VM replication only need one write step, as part of the critical path, when the VM is paused. In the case of asynchronous VM replication, we are doing it once and local, while in the synchronous case we are doing it once, but remote. There is also a second write operation which does not affect the performance of replication, but instead affects frequency of replication: asynchronous VM replication performs the transfer by copying from the local to the remote side, whereas the synchronous VM replication, as will be described in the following, requires an update of the local VM copy inside each replica. However, the latter can be delayed and performed when needed, i.e. when the primary node has crashed.

[5]Open source project supported by Open Fabrics Enterprise Distribution (OFED), http://openfabrics.org

[6]RDMA Aware Networks Programming User Manual, Mellanox, rev 1.7

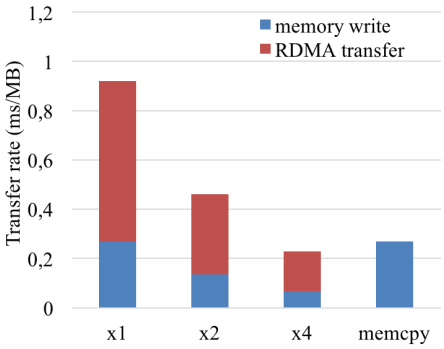

Fig. 4. Throughput of DRAM with RoCE

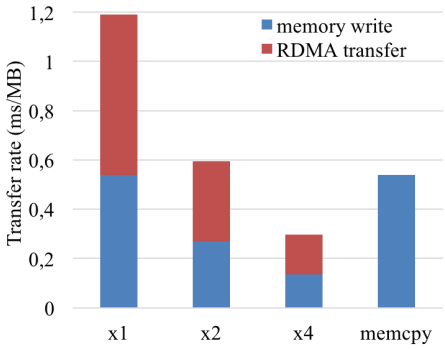

Fig. 5. Throughput of ST-MRAM with RoCE

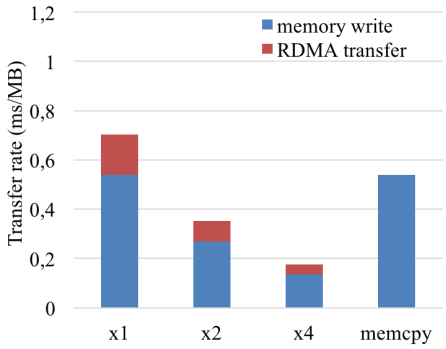

Fig. 6. Throughput of ST-MRAM with IB

B. Local copy versus distributed remote transfer

We assume that both sides utilize the same memory technology and that remote writing therefore cannot be faster than local writing. Modern network interconnects, such as RDMA, however, allow a parallel transfer of memory content via multiple network devices. Using multiple network connections as well as the earlier outlined features of RDMA, we can speed up synchronous VM replication.

As an initial analysis, we performed preliminary measurements and estimations regarding the possible speedup of synchronous VM replication when utilizing modern interconnect technology. For these measurements, we used the following infrastructure: two identical servers with 2.5GHz Intel Xeon CPU E5-2430, two ConnectX-3 Pro 10Gb PCIe dual port RoCE adapters, Ubuntu 14.04.3 LTS with Linux kernel version 3.13.0-66, and Mellanox OpenFabrics Enterprise Distribution version 3.1-1.03 as the RDMA support library. On top of this infrastructure, we compared the performance of RDMA remote transfer with local memory-to-memory copy (memcpy) operations for different sizes of data.

Our experiments demonstrate that both RDMA data transfer and $memcpy$ have linear scalability, which can be described by the function $T = A*S$ where T is the time of the transfer/write operation, S is the data size, and A is the throughput coefficient. The throughput coefficient in our case is 0.65 milliseconds per megabyte for RDMA transfer and 0.27 ms/MB for $memcpy$. As transfer size of data matters to RDMA, its overall migration time is the sum of both coefficients, while that for $memcpy$ is only affected by the second coefficient.

Fig. 4 shows the performance of RDMA transfer compared to $memcpy$. The x-axis shows different configuration schemes, where $x1$ is an original scheme without parallelization, $memcpy$ is measured in hardware as well performance of $x1$, $x2$ and $x4$ are the analytical estimations about performance in case 2 and 4 RNICs are used, respectively. It is evident that synchronous VM replication on top of DRAM and two 10GB RoCE (one device with two uplinks) cannot outperform asynchronous VM replication. Four uplinks (two devices with two uplinks) can, but the performance improvement is not significant (14.8%).

Fig. 5 shows the same experiment, but now we assume the use of ST-MRAM memory where write latency is two times higher than read latency. One can see that with slow memory, the impact of data transfer becomes more important, and while one device with two uplinks could provide near the same performance as asynchronous VM replication, it is still slower (8.5%). Two devices with four uplinks, however, will achieve a significant performance increase.

Fig. 6 shows our estimations about the use of ST-MRAM with 40GB IB. We extrapolated IB performance from 10GB RoCE by simple multiplication. The use of IB significantly decreases transfer costs, and with increased write latency, replication behavior changes extremely: parallelization of data transfer provides up to 35% performance improvement just for two uplinks, and more than 60% performance improvement for four uplinks. Furthermore, 100GB switches are on the market by now, thus even higher speedups are possible.[7]

However, these considerations are based on a simplified model of VM replication and there are some restrictions. For example, the real copying of dirty memory by replication mechanisms could be faster than the measured memcpy test because some dirty pages of the VM might be cached. Also, the division of RDMA write latency into two independent operations like transfer and write operation complicates measurements, and we therefore used approximated values which can be different for real systems. Nevertheless, it can be seen that with modern interconnects we can improve the performance of replication.

IV. Multi-site synchronous VM replication

A key element in synchronous VM replication is the data transfer from the active side to the passive one. To decrease the downtime of the replicated VM, we need to reduce the transfer time of the state changes to the remote side. We achieved this by utilizing multiple RDMA-capable network devices for parallel data transfer and identifying *dirty regions* as transfer units and for effective load balancing. To implement this, we use the following features of RDMA:

- We use multiple network devices on the server side to perform parallel data transfer. This is possible since RNICs do not block or hold registered memory regions.

[7]https://www.mellanox.com/related-docs/products/Ethernet_Switch_Brochure.pdf

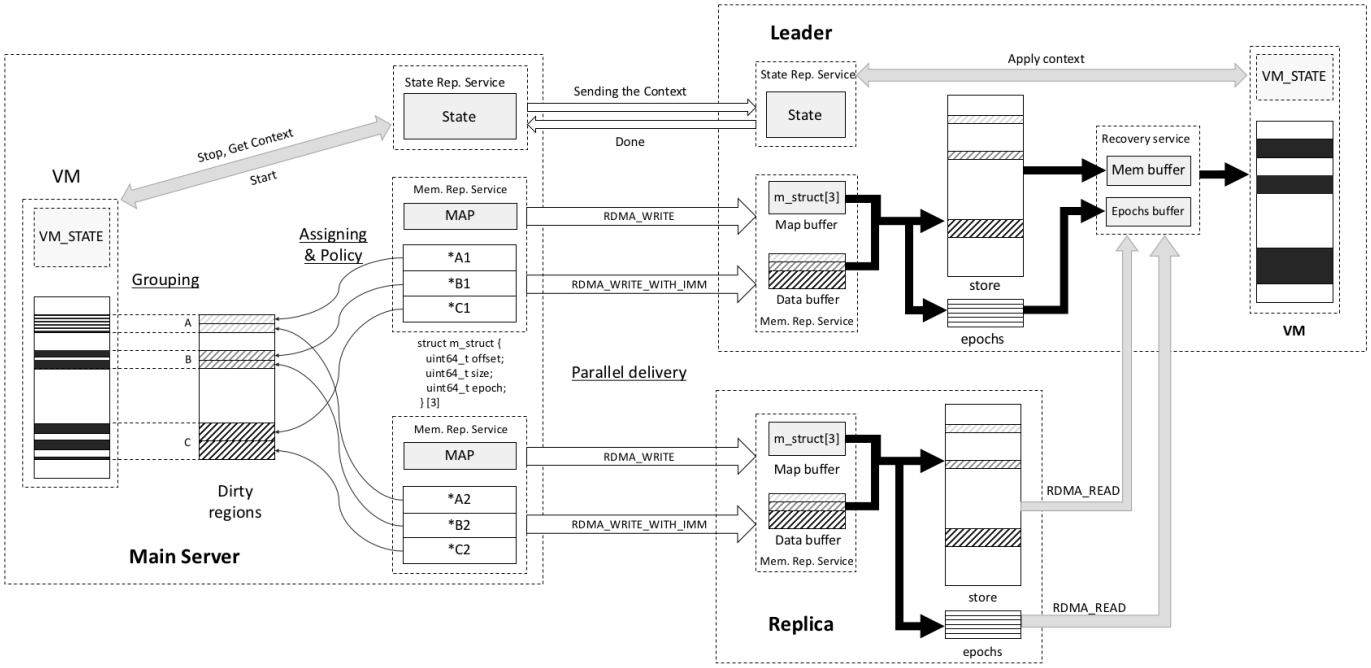

Fig. 7. Synchronous VM replication: Main Server with running VM and two replication servers

- We directly access the virtual memory of a VM without any intermediated copying, since RDMA enables direct remote transfer.
- We group modified data and transfer theses groups via SG lists.

A. Design

Fig. 7 shows the architecture of the synchronous VM replication approach. As one can see, there is the Main Server (MS) with a VM, which requires protection. The server is connected to a maximum of four Replication Servers (RSs) via RDMA-capable network cards. Futhermore, there is an ordinary TCP/IP network for sending additional information. RSs are used to store parts of the VM memory image. All RSs receive data from the MS, but one of them also receives the additional context (device states and CPU context) of the VM. We call this server the *Leader*. The Leader, like the MS, has a virtual environment with a hypervisor to host a backup VM. In case of a crash of the MS, the Leader can recover the VM using microcheckpoints. Replicas, in contrast to the Leader, only store microcheckpoints and thus do not need a hypervisor.

Our fault assumptions are as follows: our primary goal is to tolerate crashes of the MS. Should it fail, the state of the VM can be recovered from the RSs. We do not assume that the MS and parts of the backup system fail at the same time. Nevertheless, plain replicas can fail, except the Leader. However, the current architecture can be modified to be completely symmetric by broadcasting the VM state to all replicas, in which case the Leader can also fail.

The MS has a VMM with an extension, which performs continuous synchronous VM replication, thereby the following steps are performed:

1) Every n ms, the VMM stops the execution of the replicated VM
2) The VMM identifies dirty pages and prepares the list of non-dirty regions between them
3) The VMM performs grouping of dirty regions by merging the closest dirty pages
4) The VMM assigns parts of dirty regions to different RDMA devices in accordance with a user-defined policy and performs transactions in a synchronous manner. Also, at the same time, the VMM transfers the VM state via the TCP/IP connection and applies this state to the remote backup VM hosted by the Leader.
5) The VMM resumes execution of the VM for n ms. At the same time, each replica replaces the local set of dirty pages by the received new data.

B. Memory management

A VM consists of the VM state (the state of devices including CPUs) and the memory image. To recover the VM, we need to have a full set of the memory pages together composing the memory image and the VM context. As shown above, we use the Leader replica to store the backup VM and apply the backup VM state to the replicated VM. Typically the size of the context usually does not exceed hundreds of kilobytes while the size of transferred dirty pages easily exceeds 50 MB.

Obviously, we cannot send the entire VM each round. Instead, we track dirty pages, i.e. we only send changes that occurred after the previous round. For live migration [18] and asynchronous VM replication, there is usually only one remote server and all new dirty pages just are used to update outdated pages at the remote side. In our case, we use several remote servers to perform parallel writing, and the same dirty pages

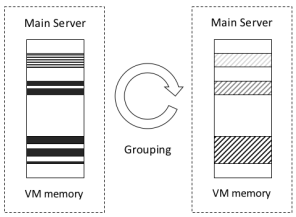

Fig. 8. Grouping of dirty pages

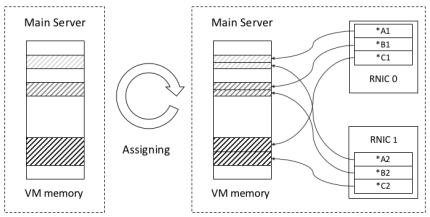

Fig. 9. Assigning of dirty regions to multiple RNICs

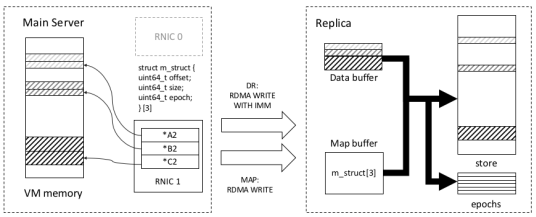

Fig. 10. Parallel delivery

from different rounds can be delivered to different remote servers because of the replication policy and our proposed balancing algorithm. Moreover, we deal with dirty regions (a contiguous set of dirty pages) and send more than one region at the same time, since RDMA provides SG support. We accompany each dirty region with a map, which describes where this memory region should be stored and from which round (called epoch number) this data came from. As a result, each replica has a copy of the replicated VM's memory represented as a set of pages from different rounds and different regions.

Fig. 8 provides a detailed example of the memory delivery mechanism. On the server side, we use the VM memory region as a source of data and one small memory region to map the transfer. On the remote side, we also allocate two buffers: one buffer for the memory transfer and one for the memory map. Moreover, on the remote side there are two additional buffers not involved in data transfer, where the replica stores pages and page epoch numbers. Regions for data and regions for the map are registered with a QP and both regions are used for RDMA transfer.

Imagine that the VMM assigned three dirty regions (gray boxes on the diagram) to be delivered via a single RNIC. Dirty regions are not contiguous and distributed across the VM memory, but after delivery, all regions will be contiguous since we send all three regions at the same time. Thus, we need to restore the original page position on the remote side in some way. For that we send the structure `m_struct`, which describes each region: offset of the region, the size of the region, and the epoch number. Firstly, we deliver this map via RDMA_WRITE without IMM notification. Then we prepare SG list for RDMA in accordance with the map and queue RDMA_WRITE with IMM. We need IMM to notify the remote side that the data transfer is complete, after which the remote side can update the local store with fresh data. This is done by a thread, thus without blocking. The server can continue the execution immediately when the transaction is completed. On the replica side the delivered data is safe, and the replica must perform an update of the local store before the next epoch comes.

An update of the local store is performed as follows: step-by-step, we take pages from the data buffer and copy them to the local store at the proper offset. We also update the epoch numbers of the corresponding pages in the epoch buffer. In case of a failure, the Leader reads all epoch and data buffers from the replicas in a similar way and then, step-by-step, copies pages with larger epoch numbers to the backup VM.

C. Load balancing, assigning, and policy

The algorithm of assigning, i.e. how parts of dirty regions are assigned to each RNIC, fulfills several important functions. Firstly, assigning balances the load: within the same period of time, each network card should deliver almost the same amount of data. If one network card sends more than others, the downtime increases since we do a blocking transfer and cannot resume execution until all RNICs complete transfer. Secondly, assigning enables fault-tolerance policies. In the moment of assigning, the VMM decides whether all replicas will receive the same data or different chunks. Since each replica receives sub-regions of dirty regions, overlapping of such regions defines the redundancy level and performance: when all replicas receive the same data, the downtime and redundancy are the highest. Similarily, when all replicas receive unique dirty regions, the redundancy as well as the downtime are the lowest.

For the "mirroring" ("fault-tolerant") mode, there is no special algorithm of assigning because each RNIC sends the same data: we prepare identical records for each dirty region in the map structures.

For the "performance" ("fast") mode, when there is no duplication of pages, we developed a balancing algorithm. We analyzed the structure of dirty regions and found several issues. The first issue is the granularity of the data transfer: we cannot transfer less than one memory page. Secondly, we observed many small dirty regions, especially with an idle VM. Another common issue is that dirty regions are not evenly divisible by the number of RNICs. Moreover, after several experiments we found that the assigning algorithm shall consider previous assignments to provide for the same load of RNICs not only with one dirty region, but within a period of time. As an example, imagine that we have three RNICs and the size of the first dirty region is 17 pages. 17 is not evenly divisible by 3, thus we need to split 17 pages as evenly as possible. For example, our algorithm splits 17 pages into [5, 6, 6]. This split is the closest to the mean value and all RNICs will send near the same number of pages, therefore there is no better division. But if we have the same task (dirty region) several times in a row, for example 5 times, the overall load processed by RNICs will be as follows: [25, 30, 30]. The difference in the load of RNICs is 16%. The situation becomes more dramatic as time goes on due to the accumulation of this mismatching. Our experiments demonstrate situations when the difference in load exceeds 40% for different RNICs.

Our heuristic to load balancing consists of the following

two components. Firstly, we split each dirty region to even and equal subregions. Secondly, we perform rotation of subregion assigning. The Algorithm 1 describes the process of assigning in pseudocode. Assume that we need to transfer a set of $regions$ dirty regions via $REPLICAS$ RNICs. Each region has $offset$ and $size$ stored in array $regions[i]$ where i is the current region. For each replica, we identify the assignment based on the number of rotations b and a uniform distribution of load defined by an even division of unassigned region size.

Algorithm 1 Integer partition

```
1:  b ← 0
2:  for i ← 0, regions do
3:      csize ← regions[i].size/4096
4:      offs ← 0
5:      for r ← 0, REPLICAS do
6:          h ← (r + b) mod REPLICAS
7:          step ← (csize)/(REPLICAS − r)
8:          maps[i][h].offset ← regions[i].offset + offs
9:          maps[i][h].size ← step ∗ 4096
10:         csize ← csize − step ∗ 4096
11:         offs ← offs + step ∗ 4096
12:         maps[h].stamp ← stamp
13:     end for
14:     b ← (b + 1) mod REPLICAS
15: end for
```

Let's take our previous example and send five dirty regions consisting of 17 dirty pages each via three RNICs. For the first region, the balancer b is 0. For the first replica, $csize$ is 17 pages, the identifier r is 0, $step$ is $17/(3-0) = 5$ pages started from $offs = 0$. For the second replica, the $csize$ is $17 - 5 = 12$, the identifier r is 1, $step$ is $12/(3-1) = 6$ pages started from $offs = 5$. For the third replica, the $csize$ is $12 - 6 = 6$, the identifier r is 2, $step$ is $6/(3-2) = 6$ pages started from $offs = 11$. For the second dirty region, the balancer b is 1, thus all assignments perform a shift to the right. The table II shows assignments for all five regions.

TABLE II
BALANCED ASSIGNMENTS

Region	R0	R1	R2
0	5	6	6
1	6	5	6
2	6	6	5
3	5	6	6
4	6	5	6
$\sum$	**28**	**28**	**29**

As one can see, all RNICs have near the mean value within each dirty region and within the entire set of dirty regions. Our experiments with real load after 1000 transactions demonstrate the imbalance in data transfer via four RNICs to be less than 2%.

Next, we will describe how we prepare dirty regions for the actual remote transfer.

D. Grouping

Grouping is an operation on dirty pages for the purpose of reducing their scatter. There are two reasons to do that: the limited size of the SG list provided by RNICs and the characteristics of dirty memory structure.

To analyze the structure of dirty memory, we compiled the Linux kernel with *kcbench* inside a VM with 512MB of virtual memory and 2 CPU cores. Fig. 11 and Fig. 12 provide statistics about the distribution of dirty and non-dirty (unmodified) pages at some random moment during kernel compilation 100 ms after the last checkpoint has been made.

As one can see, each round the VM modifies a huge number of 1-2 page sized regions separated by very small 1-2 page sized non-dirty regions. Transfer of these one-page sized regions is inefficient (we will provide a comparison of grouped and non-grouped transfers in Section V). Furthermore, our RDMA cards can only send 32 SG regions within one transaction. Thus, we decided to use grouping of dirty pages.

The algorithm for grouping consists of two phases. Since our goal is to merge dirty pages into solid dirty regions, we should prepare a set of non-dirty regions and then remove the smallest one. To achieve this, we first prepare a list of non-dirty regions, and then, in the second phase recursively remove the smallest regions until the total amount of regions does not reach the limit. The limit in our case is 32 regions, i.e. the number of SG regions supported by hardware. All operations are performed on pointers, i.e. there is no copying of pages.

After grouping the dirty pages into 32 dirty regions, the VMM is ready to apply the policy and perform assigning of subregions of dirty regions to different RNICs.

V. EVALUATION

Our evaluation was focused on the validation of the initial estimations provided in Section III. We implemented synchronous VM replication on top of qemu-kvm and used the compilation of the Linux as a benchmark in different configurations.

A. Implementation

We carried out synchronous VM replication on QEMU version 2.3.50. We took the general design of continuous replication from qemu-mc [14] and reworked it significantly in accordance with our approach. The mechanisms that we took without modifications are the following: dirty page tracking, VM state extractor and serializer as well as VM state delivery via TCP/IP. Grouping, assigning, and RDMA-based delivering of dirty regions were implemented from scratch and integrated into the qemu-mc source tree. The remote side, i.e. the replication servers were also implemented from scratch. Also, in our benchmarks we always used the qemu-mc source tree as a baseline for performance and functionality since qemu-mc was developed in a similar way to REMUS, i.e. qemu-mc provides asynchronous VM replication.

For our test bed, we used five identical servers with a 2.5GHz Intel Xeon CPU E5-2430. Four of these five servers were used as RSs and one as the MS. The MS was equipped with two ConnectX-3 Pro 10Gb PCIe dual port RoCE adapters. Also,

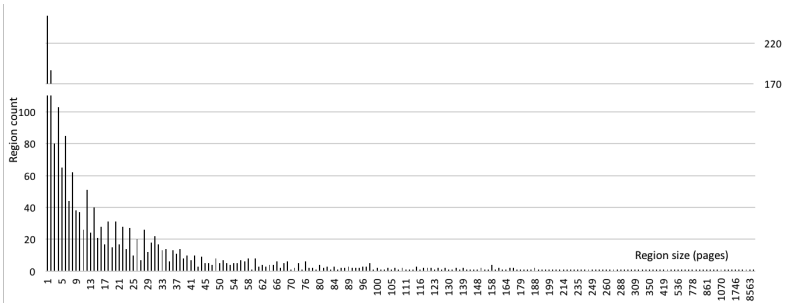

Fig. 11. Number of Non-modified regions with different size

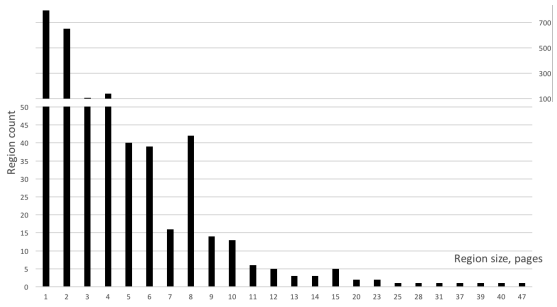

Fig. 12. Number of dirty regions with different size

each RS was equipped with the same adapter. All RSs were attached to the switch via SFP+ connectors and the MS was attached to the switch via four connectors. This connection scheme enables us to create a set of virtual networks that simulates a one-to-one connection of the MS and the RSs. In addition to RNICs, we also used a 10Gb Ethernet network for sending VM states from the MS to the Leader.

In terms of software, we used Ubuntu 14.04.3 LTS with Linux kernel version 3.13.0-66 on all servers as a host operating system, and Fedora 19 with Linux kernel version 4.1.13 as a guest VM. RDMA support was provided by Mellanox OpenFabrics Enterprise Distribution version 3.1-1.03.

B. Benchmarks

As an application workload, we used *kcbench* on a VM of 512MB with two virtual CPU cores. The MS and the Leader used the VM image provided by an independent network storage and shared via NFS, so there was no storage replication. We compared our implementation of synchronous VM replication with asynchronous VM replication provided by *qemu-mc* and native execution without continuous replication. In our experiments we used a delay of 100 ms between checkpoints, two modes of work – native execution and emulation of read/write asymmetry, two different policies ("fast" and "fault-tolerant") and different count of RNICs (2 and 4). For each test we performed three runs of kernel compilation and averaged the results.

1) Native execution and emulation of asymmetry: The overall kernel compilation time was measured for two modes. The first mode is *native* execution of the VM and replication service on existing commodity hardware. In this mode, there is no asymmetry in read/write latency. The second mode is an emulation of ST-MRAM behavior. Read latency of DRAM and ST-MRAM from Everspin are nearly the same [12], while write latency of NV-RAM is twice higher: we decided to emulate the behavior of the persistent system by slowing down write operations.

In the case of *qemu-mc*, we doubled write operations in the replication subsystem of the VMM, while in the case of synchronous VM replication we added timeouts in RSs proportional to the size of microcheckpoints. This emulation is very rough since we cannot decrease[8] memory latency on servers. Instead, we slow down some memory-related operations that, in fact, are only part of the asymmetry impact. For example, increasing write latency leads to a decrease in VM performance, which in turn in turn decreases the count of dirty pages produced within one period. This should decrease the downtime, as it depends on the size of transferred data. We assume that the increase of write latency by a factor of two leads to a proportional reduction in the number of dirty pages, and following this assumption we adjusted the timeout coefficients identified in Section III.

We also measured VM downtimes. Although the overall performance is affected by replication performance, it is not the main factor. For example, compilation time also depends on network bandwidth of the network-attached storage used by the VM. The kernel compilation time provides the integral characteristic of the virtualization system, while downtime provides precise measurements of replication implementation impact. Previously, in Section III, we made preliminary estimates of synchronous VM replication use, and now we can compare the results with the original model.

2) Measurements: Table III shows the kernel compilation times and VM downtimes for the different configurations, divided into four columns. The first column includes absolute values of the indicator time (compilation time or downtime) in seconds or microseconds. The second, third and fourth columns display the same values, but in relation to the baseline.

As one can see, *qemu* compilation, i.e. execution of kcbench without replication, requires 338 seconds. Asynchronous VM replication provided by *qemu-mc* increases the compilation time by 57 percentage points (pp). Synchronous VM replication in "fast" scheme, in turn, increases compilation time by 43 pp in four uplinks mode, and by 116 pp in two uplinks mode. The "fault-tolerant" scheme, when all RNICs send the same data, increases compilation time by up to 258 pp.

Fig. 13 and Fig. 14 visualize kernel compilation time and average VM downtime, respectively, for two configurations: native execution and ST-MRAM emulation. We can draw several conclusions from the data.

Firstly, these results confirm some preliminary estimations provided in Section III: In the case of DRAM, four uplinks demonstrate decreasing compilation time (9%), as well as decreasing average downtime (measured 60%, estimated 17%). Also, for ST-MRAM we see that four uplinks decrease

[8]Our hardware does not allow to change DRAM latency.

TABLE III
KERNEL COMPILATION TIMES AND VM DOWNTIMES FOR THE DIFFERENT CONFIGURATIONS

		Kernel compilation time				VM Downtime			
		Time, s	base=qemu, %	base=qemu-mc, %	base=fast(4), %	avg. downtime, ms	base=qemu, %	base=qemu-mc, %	base=fast(4), %
DRAM	qemu	338	**100**	64	70				
	qemu-mc	531	157	**100**	110	38	**100**	161	54
	fast(2)	731	216	138	151	37	99	159	54
	fast(4)	483	143	91	**100**	23	62	**100**	34
	ft(4)	1211	358	228	251	69	184	295	**100**
MRAM	qemu-mc	782	**100**	84	127	60	**100**	114	181
	fast(2)	927	119	**100**	151	52	88	**100**	158
	fast(4)	614	79	66	**100**	33	55	63	**100**
	ft(4)	2009	257	217	327	146	244	278	440

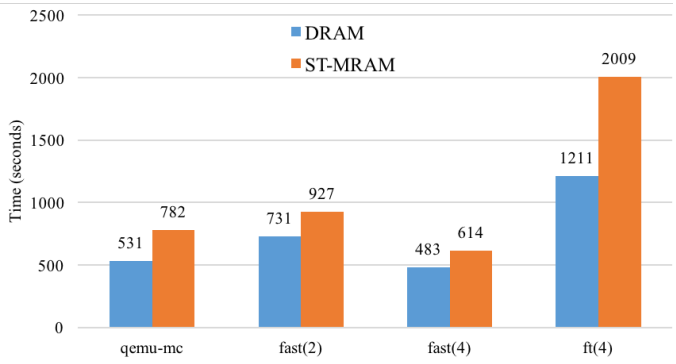

Fig. 13. Average value of kernel compilation time

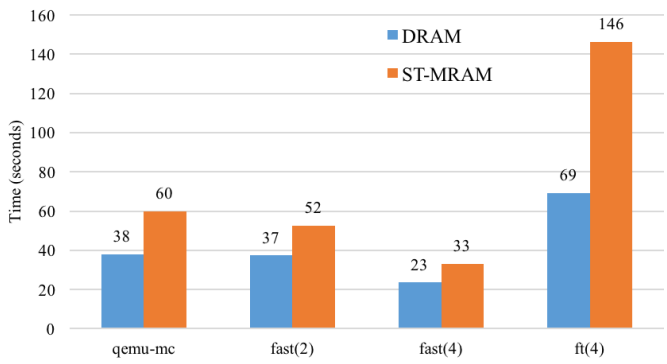

Fig. 14. Average VM downtime

downtime ultimately as well as decreasing compilation time.

Secondly, we see that there is a mismatch between modeled and measured data in emulation of ST-MRAM and DRAM. For example, "fast(2)" mode in our model should provide near the same downtime value as "qemu-mc" in the case of ST-MRAM use and a 1.5 downtime increase in the case of DRAM use. But we do not see such behavior: In DRAM, "fast(2)" demonstrates near the same downtime value as "qemu-mc", and with ST-MRAM emulation, "fast(2)" is better than "qemu-mc" by 13 pp. Thus, measured average downtime values are better than modeled. Meanwhile, compilation time as well as overall performance are closer to modeled behavior: "fast(2)" takes more time than "qemu-mc". This proves that overall performance does not only depend on downtime.

3) Recovery time: The recovery process is initiated and performed by the Leader. The Leader tracks the RDMA connection and detects a *on_disconnect* event from the QP. Also, recovery is initiated if there is no incoming data from the MS within the user-defined timeout. The Leader has the actual state of the VM since the MS sends its actual state each replication. Since the Leader does not have the actual memory image, the recovery time is the time needed for reading memory from replicas to the Leader.

Recovery has a constant time depending on number of replicas and the size of the VM. The structure of local sets, i.e. the number of dirty pages inside each replica is irrelevant since we read the whole memory region. In our experiments, the average recovery time for the 512MB VM with four replicas did not exceed 2.0s (the average value is 1.94s).

4) Measurement of overhead: Fig. 11 and Fig. 12 show the distribution of dirty pages in the VM. As mentioned before, during the process of kernel compilation, there are many small dirty regions interleaved with non-dirty regions. Our mechanism of grouping removes some of the non-dirty regions, and as a result, there is an overhead, since RNICs should also transfer non-modified pages. For each transaction, we compute the overhead as a ratio of transferred page count to dirty page count.

For the kernel compilation, the overhead values differ, depending on the compilation phase and the replication intervals. For example, for a delay of 500 ms, the overhead varies from 0% to 2304% with the average value 513%. For a 100 ms delay, the overhead range is more extensive: from 186% to 14163% with the average value 813%. At first sight, these overhead values seem immense, but most of them are the result of small transfers (tens of kilobytes).

After performing these experiments, we can draw several conclusions. Firstly, high-frequency microcheckpoints have fewer, but more scattered dirty pages, whereas low-frequency microcheckpoints have more, but less scattered dirty pages. Secondly, any mechanisms for increasing the dirty pages generation rate, for example, by increasing the number of compilation threads, also makes dirty pages less scattered.

In other words, the replication performance significantly depends on the content of the guest VM, while the location of dirty pages depends on the program and allocation mechanism provided by the operating system. General-purpose operating systems like GNU/Linux provide an universal allocator and

do not take migration problems into account. We think that specialized projects like OSv [19], developed especially for the virtual environment, could be a good platform for the development of a migration-aware memory allocator.

5) Sequential sending versus grouping: With 813%, the average overhead resulting from grouping looks huge. We decided to compare grouped delivery with *sequential delivery*. Sequential delivery is a RDMA write of all dirty regions without grouping. Sequential delivery cannot be performed as a single transaction as we made it with the grouped one, but we used all 32 SG for each delivery to increase performance. The results were surprising: *kcbench* demonstrates longer compilation time as compared to delivery with grouping, with a difference of 8-12 seconds. The reason for this is the structure of dirty memory: as we showed previously, each microcheckpoint consists of a large number of small dirty regions shuffled with small non-dirty regions, and sometimes delivery of one large region is much faster than sequential delivery of many small regions.

VI. Related works

This work revisits virtual machine replication in the light of novel hardware trends. Remus [11], Kemari [13], qemu-mc [14], as well as commercial products like VMware [20] focus on an asynchronous VM replication of changes. There are several extensions to this general concept, like an implementation of copy-on-write techniques [21], [22] to reduce downtime. In contrast to these approaches, we propose to transfer dirty pages synchronously in a zero-copy manner and parallelize the transfer for performance improvements.

Huang et al. utilize RDMA to speed-up the migration of VMs [23]. They propose their own mechanism for memory registration and provide an approach for non-contiguous data transfer. In contrast, we always copy memory directly host-to-host aided by a grouping of pages instead of remapping them. Furthermore, our solution supports different HA policies.

VII. Conclusion

Non-volatile memory technologies are characterized by strong asymmetry of read and write access times. In this work, we demonstrate how a common HA practice in a rack-scale environments – asynchronous VM replication – needs to be revised when facing asymmetric read/write access times. We proposed and implemented *multi-site synchronous VM replication*, which performs contiguous migration of VMs in a zero-copy manner. The proposed synchronous VM replication utilizes different features of RDMA, like parallel reading of virtual memory by multiple RNICs, offloading network communication from the OS and the smart use of hardware supported SG lists. Effective load balancing by grouping and parallelly transferring dirty pages increases the replication performance compared to common asynchronous VM replication by 10% in a commodity DRAM-based system, and up to 27% for emulated prospective NV-RAM-based systems.

References

[1] K. Asanovic and D. Patterson, "Firebox: A hardware building block for 2020 warehouse-scale computers," in *USENIX FAST*, vol. 13, 2014.

[2] M. Wuttig, "Phase-change materials: towards a universal memory?" *Nature materials*, vol. 4, no. 4, pp. 265–266, 2005.

[3] D. Narayanan and O. Hodson, "Whole-system persistence," in *ACM SIGARCH Computer Architecture News*, vol. 40, 2012.

[4] V. A. Sartakov and R. Kapitza, "NV-Hypervisor: Hypervisor-based Persistence for Virtual Machines," in *Workshop on Dependability of Clouds, Data Centers and Virtual Machine Technology (DCDV 2014)*, 2014. [Online]. Available: https://www.ibr.cs.tu-bs.de/users/sartakov/papers/sartakov14dcdv.pdf

[5] R. F. Freitas and W. W. Wilcke, "Storage-class memory: The next storage system technology," *IBM Journal of Research and Development*, vol. 52, no. 4/5, p. 439, 2008.

[6] J. C. Mogul, E. Argollo, M. A. Shah, and P. Faraboschi, "Operating system support for nvm+ dram hybrid main memory." in *HotOS*, 2009.

[7] G. W. Burr, B. N. Kurdi, J. C. Scott, C. H. Lam, K. Gopalakrishnan, and R. S. Shenoy, "Overview of candidate device technologies for storage-class memory," *IBM Journal of Research and Development*, vol. 52, no. 4.5, pp. 449–464, 2008.

[8] M. K. Qureshi, M. M. Franceschini, A. Jagmohan, and L. A. Lastras, "Preset: improving performance of phase change memories by exploiting asymmetry in write times," *ACM SIGARCH Computer Architecture News*, vol. 40, no. 3, pp. 380–391, 2012.

[9] S. Schechter, G. H. Loh, K. Strauss, and D. Burger, "Use ecp, not ecc, for hard failures in resistive memories," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3. ACM, 2010, pp. 141–152.

[10] R. Azevedo, J. D. Davis, K. Strauss, P. Gopalan, M. Manasse, and S. Yekhanin, "Zombie memory: Extending memory lifetime by reviving dead blocks," in *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3. ACM, 2013, pp. 452–463.

[11] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield, "Remus: High availability via asynchronous virtual machine replication," in *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, 2008, pp. 161–174.

[12] Everspin, "Mram into mainstream," 2016.

[13] Y. Tamura, "Kemari: Virtual machine synchronization for fault tolerance using domt," *Xen Summit*, vol. 2008, 2008.

[14] M. R. Hines. (2013) Rdma migration and rdma fault tolerance for qemu. [Online]. Available: http://www.linux-kvm.org/images/0/09/Kvm-forum-2013-rdma.pdf

[15] *288pin Registered DIMM based on 8Gb B-die*, Samsung, 2015, rev. 1.3.

[16] R. Recio, B. Metzler, P. Culley, J. Hilland, and D. Garcia, "A remote direct memory access protocol specification," RFC 5040, October, Tech. Rep., 2007.

[17] S. Bailey and T. Talpey, "The architecture of direct data placement (ddp) and remote direct memory access (rdma) on internet protocols," *Architecture*, 2005.

[18] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*. USENIX Association, 2005, pp. 273–286.

[19] A. Kivity, D. Laor, G. Costa, P. Enberg, N. Har'El, D. Marti, and V. Zolotarov, "Osv—optimizing the operating system for virtual machines," in *2014 usenix annual technical conference (usenix atc 14)*, 2014, pp. 61–72.

[20] D. J. Scales, M. Nelson, and G. Venkitachalam, "The design of a practical system for fault-tolerant virtual machines," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 4, pp. 30–39, 2010.

[21] M. H. Sun and D. M. Blough, "Fast, lightweight virtual machine checkpointing," 2010.

[22] B. Gerofi and Y. Ishikawa, "Rdma based replication of multiprocessor virtual machines over high-performance interconnects," in *2011 IEEE International Conference on Cluster Computing*. IEEE, 2011, pp. 35–44.

[23] W. Huang, Q. Gao, J. Liu, and D. K. Panda, "High performance virtual machine migration with rdma over modern interconnects," in *2007 IEEE International Conference on Cluster Computing*. IEEE, 2007, pp. 11–20.