SmartNIC-enabled Live Migration for Storage-Optimized VMs with Pyrocumulus

Jiechen Zhao[♥♣], Ran Shu[♣], Lei Qu[♣], Ziyue Yang[♣], Rui Ma[♣],
Derek Chiou[◊♠], Natalie Enright Jerger[♥], Peng Cheng[♣], Yongqiang Xiong[♣]
[♥] University of Toronto, [♣] Microsoft Research Asia, [♠] UT Austin, [◊] Microsoft

Abstract

Cloud providers offer storage-optimized VMs equipped with locally attached storage to meet the high performance requirements of cloud users. Live migration (LM) is crucial for such VMs to improve availability and manageability. However, providers do not enable LM for storage-optimized VMs. Host-managed LM suffers from high resource overheads and varied user performance, while offloading LM to SoC SmartNICs or disks cannot reliably accomplish LM in a reasonable time. The fundamental challenges are (1) consistency, demanding a high resource budget, and (2) network contention, preventing LM from converging. We propose Pyrocumulus, an LM approach on FPGA SmartNICs, enabling SLA-aware, fast, and low-overhead LM for storage-optimized VMs. We exploit hardware customizability and efficient network accessibility of the FPGA SmartNIC with LM protocol, architecture, and algorithm designs. Results from our FPGA SmartNIC prototype show that Pyrocumulus reduces user latency variances during LM up to 12.4×, lowers LM time by up to 19.6×, and saves cost up to 3×, while only taking 0.9%/3.8% compute/memory overhead of a mid-end FPGA SmartNIC.

1 Introduction

Many large-scale online services rely heavily on storage performance [8, 35]. Tenants have specific performance requirements when building their storage applications. To meet those requirements, cloud providers offer *storage-optimized VMs* [21, 22]. Such VMs have high speed NVMe SSDs directly mapped to VMs to provide millions of IOPS per VM. Many services, e.g., Databricks and Microsoft Data Explorer, have already benefited from storage-optimized VMs [55].

Meanwhile, *live migration* (LM) is a key technique to guarantee minimal service interruptions during management tasks, including system updates, configuration changes, hardware failure handling, and allocation optimizations [13, 14]. Although the total LM time can be up to hours, VMs only need to be temporarily paused as low as sub-second levels [62]. There are several *benefits* of enabling LM. First, it significantly alleviates minute-level service interruptions [11]. Second, it improves the availability of VMs by up to 50% [31], thus satisfying more SLAs. Third, it improves the flexibility of large-scale maintenance while satisfying SLAs.

However, LM is *not currently supported* for storage-optimized VMs [13, 22]. The root cause is that existing solutions cannot achieve affordable resource overhead and high LM performance simultaneously. A successful LM requires sufficient (1) compute and memory resources to track data consistency as the source disk can be read or written by the VM during LM, (2) network resources to migrate data and keep source and destination disks consistent, and ultimately converged. A huge amount of compute resources should be statically provisioned[1] for an LM to avoid resource exhaustion. Networks should also not become the bottleneck or the source of interference; otherwise, LM may never finish, leading to undesirable unpredictability of user performance.

An existing solution, the Host Controlled approach, uses the hypervisor to manage LM [3–5, 25, 30, 54, 62]. However, this approach requires too many resources provisioned permanently, up to tens of CPU cores, due to high software overhead during LM. Moreover, the performance of the VM fluctuates due to CPU and network resource contention with the LM. An alternative solution to address user interference is to offload LM to SoC SmartNICs. However, the LM time becomes unacceptably long, up to hours or even not converging, due to the SoC's limited computational capability [42, 52]. Because LM should be statically provisioned, the SoC has little to no resources left to offload other service features [9, 15, 42, 49]. Recent NVMe SSD advances can offload consistency tracking to SSDs [26, 27]. However, resource overhead is still high because the CPU needs to process logs made by disks. The common double writes algorithm for storage migration [62] is not supported, whereas tracking and updating dirty pages takes up to 17.8× longer LM time.

[1] Alternatively, an on-demand way introduces contention, which may lead to performance interference and user dissatisfaction.

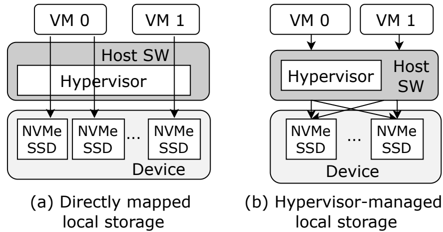


Figure 1: Different local storage systems in the cloud.

We propose to *enable LM for storage-optimized VMs by provisioning hardware resources on FPGA-based SmartNICs for LM*. The goals of PYROCUMULUS are threefold: minimizing (1) user interference during LM, (2) LM time, and (3) LM resource overhead. First, PYROCUMULUS statically provisions a minimal set of compute and memory resources for LM on the FPGA, eliminating host resource contention between the LM and the VM, thus reducing user latency variances by up to 12.4×. Second, by taking advantage of SmartNIC's efficient network accessibility and FPGA's customizability, we eliminate the software overhead and reduce LM time by up to 19.6×. Third, by exploiting hardware parallelism, PYROCUMULUS realizes fast LM at low resource overhead, taking only 0.9%/3.8% compute/memory resources of a mid-end FPGA SmartNIC, saving cost by up to 3×.

Following the above design principles, we first provide a system architecture and a SmartNIC-driven LM protocol (Sec. 4). The FPGA SmartNIC conducts a hardware-friendly LM algorithm with a hardware module, the LM controller. The LM controller mediates user accesses during LM to perform efficient consistency tracking based on the double writes algorithm (Sec. 5). The low-overhead LM controller only requires four hardware pointers, small control logic, and ~256KB of on-chip BRAM. To avoid LM-related network contention, PYROCUMULUS augments the LM controller with a backpressure mechanism. This is designed to provide direct control of user performance during LM. Instead of pausing the VM or canceling LM as prior work proposes [54, 62], PYROCUMULUS proposes to properly pause the LM by the LM controller when users' tail latency becomes too high. We provide guidelines on when to pause/resume LM, how to handle worst case via reshaping consistency traffic, as well as a hardware-friendly backpressure algorithm when a hybrid mode when LM is paused and unpaused (Sec. 6).

We implement our prototype on Intel Arria 10 FPGA based SmartNICs connecting over 100Gbps network. We study various workload and traffic patterns in terms of read/write ratio, request rate, and access patterns. Our prototype also tests the LM of a VM running a RocksDB application, with only a latency increase of 1.4% in 99.99th percentile latency.

2 Background

Three types of block storage are available for VMs in clouds.

Persistent block storage. It stores data in remote storage clusters and emulates a block interface on the host for VMs. It provides high durability and availability for customers. Examples of persistent storage are AWS EBS [12], Azure Managed Disks [10], and Google Persistent Disk [20]. Although it provides many benefits, it does not offer high performance. Thus, cloud providers also offer two *local storage* options, which are introduced next.

Directly mapped local storage. To provide high capacity and high IOPS on a single VM, providers offer directly mapped NVMe disks (both SSDs and HDDs) to VMs. This type of VM is called ***storage-optimized VM*** [21, 22]. Fig. 1(a) illustrates its system architecture. These local disks are not managed by the provider's hypervisor, but by users themselves, and the entire disk(s) is directly mapped to those VMs and not shareable with other tenants. A VM can have direct access to more than one NVMe SSD.

Hypervisor managed local storage. Providers use hypervisors to acquire control of local storage, as shown in Fig. 1(b). Providers use this architecture to manage many types of compute VMs with local storage [23]. Note that those VMs are not storage-optimized. The hypervisor-level management usually introduces a non-trivial amount of overhead, which is proportional to overall I/O activity [45].

2.1 Live Migration Algorithm

In practice, cloud providers predominantly adopt pre-copy algorithms [14, 24, 62]. where the hypervisor first copies the VM's current state and then iteratively transfers deltas. The source disk remains continuously updated and accessible to the VM, regardless of whether LM is active. Fig. 2 shows a pre-copy workflow. An alternative is post-copy [39], where the VM is first relocated and missing data blocks are subsequently fetched from the source on demand. It is usually complex [54]–data is lost if the network fails, and resources on the source cannot be released until both sides fully converge.

Defining a successful LM. First, data consistency should be correctly tracked during LM, ensuring (1) users always read the most updated version of data, and (2) the VM image resumed on the destination node has the newest updates. To maintain consistency, the dirty page tracking algorithm [25, 27, 30, 38] relies on a bitmap to iteratively copy dirty pages to the destination, whereas the double writes algorithm [54, 62] employs a non-iterative, single-pass policy, committing new writes to both nodes on each user write before moving the LM progress forward. Second, LM needs to converge within a reasonable time. Initially, the provider starts a data copy from the source to the destination. The VM can modify data on the source disk. Once the source and destination states converge—or when triggered by the provider's policy—the VM is paused, its state is transferred, and execution is resumed at the destination.

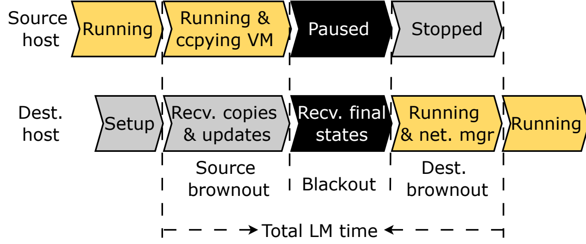


Figure 2: The pre-copy LM process between a source and a destination host when migrating a storage-optimized VM.

2.2 Rationale of Live Migration System Design

A successful LM mandates high software overhead. Since storage-optimized VMs bypass the hypervisor on the datapath, it is necessary to allocate compute resources to run a software mediation layer for consistency tracking. The mediation layer requires a minimum resource budget to guarantee successful LM convergence. Allocating fewer resources than this threshold can lead to unstable performance, prolonged convergence duration, or LM convergence failure from resource exhaustion. Software overhead is non-trivial for a successful LM. We study that we should statically provision tens of CPU cores for LM, given typical storage-intensive workload patterns.

We must statically provision compute and memory resources for LM. There are two common strategies to allocate compute/memory resources: on-demand vs. static provisioning. An on-demand provisioning strategy degrades VM performance during LM. If all CPU cores are allocated to VMs or consumed by the hypervisor, no cores remain available for LM, forcing the provider to forgo the migration. Alternatively, if the provider mandates LM, CPU cores must be reclaimed from user VMs or the hypervisor, disrupting their execution. Memory resources pose an even greater challenge: unlike CPU cores, they are far less elastic to (de)allocate, particularly when the VM sustains high user traffic during LM. This work demonstrates that, although LMs occur infrequently, providers must statically reserve sufficient resources to ensure that an LM can complete promptly.

Adaptive LM-aware network management is missing. Prior work has limited design exploration regarding the network resource management when LM is running. Adaptively sharing the network among the following co-existing traffic classes is crucial.

- **VM network traffic**: Storage-optimized VMs use their virtual NIC for communications such as remote storage accesses and inter-VM data transfers.
- **LM copy traffic**: Data blocks are transferred sequentially from the source to the destination during LM.
- **LM consistency traffic**: On each user write, an extra block-level remote write is triggered for maintaining data consistency between the source and the destination. It leads to **latency spikes for local user SSD access**. User access to a certain block cannot proceed before the remote write completes.[2] It also adds **network bandwidth contention**. If the volume of consistency traffic is too high, LM usually cannot converge and thus lasts for too long.
- **Other traffic**, such as data transfers by other VMs on this machine, and infrastructure-managed core network traffic.

Table 1: Comparisons of potential LM solutions for storage-optimized VMs. ◐ means partially achieving the goal.

Goal	Host Controlled	SoC Offload	Disk Assisted	Pyrocumulus
Transparency	×	✔	◐	✔
Low LM time	✔	×	×	✔
Low LM cost	×	×	×	✔

3 Motivation

This work aims to satisfy the following three goals.

- **Goal 1: User performance transparency during live migration (LM)**, where VM's local SSD access tail latency should satisfy SLA during LM.
- **Goal 2: Low LM time**, reducing LM duration to avoid long resource occupancy on both the source and the destination nodes, which makes the system more robust with less time occupied by high-overhead LM.
- **Goal 3: Low cost**, minimizing provisioned LM resources.

There are three baseline architectures for storage-optimized VMs to enable LM. Comparisons of goals are listed in Table 1.

- **Host Controlled**: Providers can provision sufficient CPU resources and run a piece of software to manage LM [3–5].
- **SoC Offload**: A plethora of I/O features in the hypervisor are offloaded to SoC-based SmartNICs to reduce CPU contention and save CPU cores [9, 15, 42, 49]. Providers can also provision a portion of resources on the SoC SmartNICs to offload the LM feature.
- **Disk Assisted**: Recently, Samsung has proposed hardware assistance for LM within NVMe SSDs [26, 27]. The SSD logs modifications of pages (i.e., dirty pages) during LM. The on-SSD hardware support works together with the software on the host CPU to accomplish LM.

Host Controlled: High User Interference and High Cost. The Host Controlled approach induces latency variance on other programs running on the machine due to LM's high software overhead. We co-run LM and an in-memory low-latency key-value store application, MICA [51]. In this case, the interference on the host is entirely due to data copy traffic,

[2] Normal reasons include avoiding write-after-read, write-after-write, or packet loss problems, which may crash the VM image when resuming the VM on the destination host due to correctness issues.

Table 2: MICA key-value store latency experiments w/ and w/o LM co-running. We report the maximum request latency in terms of the number of CPU cycles.

	w/o LM running	w/ LM running	Latency increase ratio caused by LM
MICA SET	0.6K	7.4K	12.3×
MICA GET	0.5K	4.9K	9.8×

and no consistency traffic is initiated because MICA does not generate reads/writes to local storage. The LM is running based on QEMU/KVM `libvirt`.[3] Table 2 lists the maximal per-request latency among all MICA SET and GET requests. Compared to standalone MICA execution (i.e., without LM), co-locating MICA with LM increases the latency of SET and GET operations by 12.3× and 9.83× due to host contention, including for shared caches and memory bandwidth.

The Host Controlled approach can enable a successful LM for storage-optimized VMs, at the cost of statically provisioning tens of CPU cores. Overall, the number of CPU cores that must be statically provisioned for a successful LM can reach as high as 15. Each Xeon core generates a revenue of $4,500 in its lifetime [36]. Thus, statistically provisioning multiple cores for LM is not economically beneficial.

Offload to SoC SmartNIC: High LM Cost or Unsuccessful LM. The SoC Offload approach incurs the same software overheads in data copying and consistency tracking as the Host-Controlled approach. Its primary limitation lies in the constrained computational capacity of existing SoC SmartNICs, which rely on relatively weak cores to handle high overhead LM. For example, the Broadcom Stingray SoC is equipped with only 8 ARM-A72 embedded cores and 16 MB of shared cache [6]. As a result, running LM on the SoC quickly overloads the device. Even when dedicating the entire SoC SmartNIC exclusively to LM—sacrificing other NIC functionalities that also depend on SoC resources—the LM time remains significantly higher than under the Host-Controlled approach. Moreover, such dedication indirectly increases costs due to disabling other offloaded features of the SoC. Hence, the SoC Offload approach is impractical.

Offload to SSD: High LM Time or Unsuccessful LM. One might think the Disk Assisted LM approach can alleviate the tension of this trade-off, in that it offloads a portion of consistency tracking overheads to disks. However, we find that although this approach works well for traditional compute VMs, the LM time is unacceptably long for storage-optimized VMs. When running LM with a 50%/50% random read/write user workload, the LM time is 3× longer than the Host Controlled approach, even at a low random write rate (50MB/s). The LM time becomes even worse when the rate of user writes increases. The root cause is that the datapath used in the Disk Assisted approach is inefficient for storage-optimized VMs with high read/write intensity. While the SSD logs modifications, the software on the CPU must process these logs. VM's writes are offloaded, where writes go directly to the source SSD controller. The software requires access to the data to track dirtied blocks. However, because the write is offloaded and bypasses the hypervisor, it cannot obtain the data directly from the VM's memory buffer. Instead, the software must explicitly read the newly written data back from the source SSD. This offloaded write introduces severe interference and resource inefficiency, as each remote write incurs an additional read operation. For write-intensive VMs, the resulting log volume that the software must process becomes enormous.

[3] This LM process migrates a 4 GB VM image over a 100Gbps network. Source and destination SSDs are both Intel Optane 900P. We set the MICA instance with 28 threads, 50%/50% GET/SET, and leave the remaining 4 threads for KVM (hyperthreading is off).

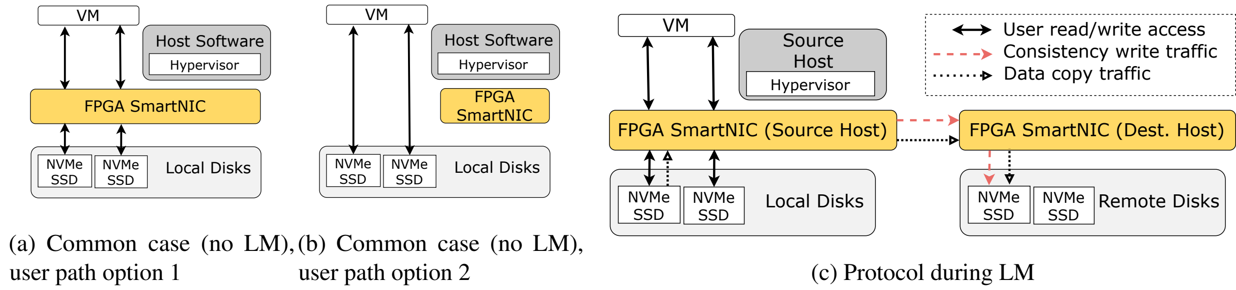
4 Pyrocumulus System Architecture

This section proposes Pyrocumulus, a system architecture design on each server that hosts storage-optimized VMs. Pyrocumulus statically provisions a small hardware design on the FPGA-based SmartNIC for LM. The static provisioning enables LM capability anytime without contending for CPU resources with VMs.

Pyrocumulus protocol. Fig. 3(a) and Fig. 3(b) show two optional data paths that Pyrocumulus supports in the common case, where no LM happens. This system architecture allows SmartNIC to intercept all I/Os from the VM, as illustrated in Fig. 3(a). Optionally, the VM accesses can also go through the native data path of a directly-mapped local storage, where both the host software and the SmartNIC are not on the critical path, as depicted in Fig. 3(b). The provider can configure the data path to be either SmartNIC-mediated (Fig. 3(a)) or native (Fig. 3(b)).

During LM, the FPGA SmartNIC works as the hardware-based hypervisor, intercepting all I/Os from the VM and coordinating them with the LM I/Os for data consistency. LM I/Os are directly initialized by the SmartNIC on the source host, entirely bypassing the CPU on the source host. Fig. 3(c) shows that the SmartNIC takes advantage of its network accessibility and directly handles the data transfers between two SmartNICs on the source and destination hosts.

Pyrocumulus architecture. Fig. 4 provides a system overview of our on-SmartNIC LM approach. The LM controller module on the SmartNIC is responsible for triggering, monitoring, and managing an LM process on behalf of the software hypervisor running on the host. There are four possible datapaths. For *VM access* (path ① in Fig. 4), we adopt the access flow introduced by FVM [45] between VMs and NVMe disks, where each NVMe command is mediated by the FPGA before going into the NVMe controller. This design eliminates host involvement. The SmartNIC on the source node handles two LM-related paths, i.e., *LM double write* ② and *LM send*



(a) Common case (no LM), user path option 1

(b) Common case (no LM), user path option 2

(c) Protocol during LM

Figure 3: The protocol overview of Pyrocumulus.

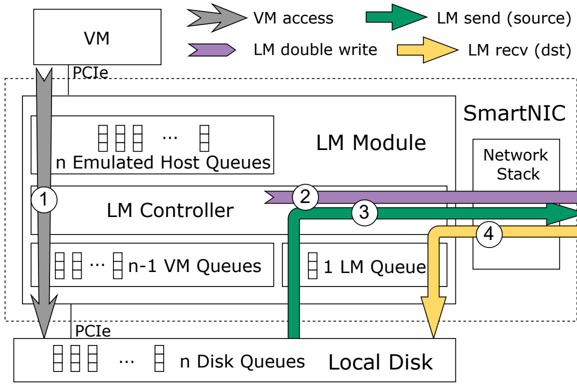


Figure 4: Architecture diagram of FPGA-based SmartNIC.

③. The destination SmartNIC handles *LM receive* path ④. Paths ①–③ exist in the source SmartNIC and path ④ exists in the destination SmartNIC.

Live migration controller. The controller proactively generates LM-related send/receive commands and talks to its local disks through the direct access interface introduced above. During LM, the controller has full visibility over commands on all paths shown in Fig. 4. Thus, the controller has full interposition for consistency among VM-SSD paths and inter-SmartNIC-SmartNIC network paths. All LM functionalities in this work are designed in the LM controller.

Hardware requirements of Pyrocumulus architecture. Pyrocumulus requires the NIC with a high-speed network, e.g., 100+ Gbps. Pyrocumulus has very low FPGA memory/logic requirements, so that nearly all FPGAs meet our requirements. We prototype our design on a mid-range FPGA, i.e., Intel Arria 10 GX 160 [1] produced more than 5 years ago. An FPGA with higher network bandwidth can maximize the benefits of Pyrocumulus.

Software requirements of Pyrocumulus architecture. Each VM installs a Pyrocumulus frontend driver that handles the datapath of accessing NVMe, bypassing the host hypervisor for high efficiency. To maximize the parallelism of the FPGA-based SmartNIC and many local SSDs, the emulated host queues on the LM module of the SmartNIC can be arbitrarily used for the VM's disk accesses. Users can specify their own I/O scheduling policies across those emulated queues. In other words, incorporating Pyrocumulus does not require changing users' scheduling routine, thus avoiding unexpected performance loss, which affects user satisfaction. While FPGA SmartNICs offload some critical and costly features, such as LM, from the hypervisor, the hypervisor still exists and runs in the host software. Within the hypervisor software, a Pyrocumulus backend driver handles control plane configurations to the SmartNIC and local storage.

5 Migration Algorithm on FPGA SmartNIC

LM requires strong consistency, reflecting the most recent writes from user programs in a VM during LM. All previous consistency tracking methods are designed for hypervisor software. They are complex to implement and tune, even in software. For example, dirty page tracking needs tremendous engineering efforts to tune parameters to make the LM finally converge, and double writes need complex code to guarantee atomicity [54]. In Pyrocumulus, consistency tracking is executed by the hardware LM controller instead of by host software. Thus, the tracking is resource-constrained and should be hardware-friendly. We use double writes because of its superior performance for VMs with local SSD [54].

Pointer-based sub-region partitions. A VM image is distributed across one or multiple contiguous address regions. Each region is defined as a contiguous logical block addressing (LBA) region. Pyrocumulus proposes a double write algorithm that is FPGA-friendly. This work partitions the LBA space of each SSD that is being migrated into three logical sub-regions, as shown in Fig. 5.

- `Migrated`, the sub-region that has finished being copied,
- `inflightMigrating`, with in-flight migration requests,
- `toBeMigrated`, the sub-region to be copied.

Pointer-based tracking with low hardware overhead. We only need 4 hardware pointers to track LM progress. This tracking also needs simple switch logic, which has low hardware overhead, to check the address of each I/O, determin-

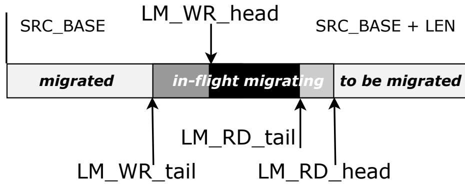


Figure 5: A snapshot of PYROCUMULUS double write LM consistency tracking with four hardware pointers.

ing which of the above three regions the I/O falls into. The 4 pointers are `LM_RD_head`, `LM_WR_head`, `LM_RD_tail`, and `LM_WR_tail`, depicted by Fig. 5. `LM_RD_head` and `LM_WR_head` indicate the progress of LM (local) read and (remote) write commands submitted, while `LM_RD_tail` and `LM_WR_tail` move forward when commands commit.

Pointer movement policy. Each of the 4 pointers can only shift right sequentially (e.g., pointer X becoming $X{+}N$). PYROCUMULUS disallows any pointer jumps for low hardware overhead. This significantly reduces the hardware logic of tracking out-of-order (OoO) scenarios, where remote writes to the destination disk may commit in an OoO manner due to inevitable factors like network congestion or remote disk latency variance for different blocks. This policy also significantly reduces the buffers needed to keep intermediate state. Note that each region defined in PYROCUMULUS is a contiguous physical address region, so pointer management does not need to handle cross-region progress tracking.

Overall live migration algorithm in PYROCUMULUS. The proposed LM algorithm resides in the LM controller on the SmartNIC in PYROCUMULUS systems, described in Alg. 1. Alg. 1 has a source node part and a destination node part, where the logic is running in the corresponding FPGA SmartNIC hardware. Alg. 1's presentation is simplified with only `LM_head` and `LM_tail`, representing `inflightMigrating` sub-region that is locked at a given time. In reality, all 4 pointers enable better pipeline parallelism on the FPGA. While LM is not complete (`LM_tail` < VM image size `LEN`) and the number of in-flight copies does not exceed the locking window size $WINDOW_SIZE$, we continue to run the loop shown in `line3`-`line6` in the LM controller hardware.

- If the address of a user I/O falls into `toBeMigrated` sub-region, i.e., the address of the user I/O is larger than `LM_head`, the LM controller directly accesses the local disk because the local disk has the most up-to-date data.
- If the address of a user I/O falls into `Migrated` sub-region, when I/Os are user writes, the LM controller uses the function `SendDoubleWrites` in `line10` to conduct the following three actions: (1) forwards the user write I/O to the local disk on the source host, (2) duplicates the user write I/O and initializes the consistency traffic to the `dest IP`, and (3) waits until both local and remote writes are successfully committed. Note that `SendDoubleWrites` is

Algorithm 1: PYROCUMULUS LM algorithm.

```
   // In source SmartNIC:
1  Procedure Live Migration():
2    ResetPointers();
3    while LM_Tail < LEN and
       Num_inflight < WINDOW_SIZE do
4      data ← ReadLocalDisk(LM_Head);
5      SendCopyData(dest IP, LM_Head, data,
         FLAG_COPY);
6      LM_Head ← LM_Head + 1;
7  Function OnTenantWrite(address, data):
8    if address < LM_Tail then
9      if no inflight writes on address then
10       SendDoubleWrites(dest IP, address, data,
           FLAG_DOUBLE_WR);
11   else
12     if address > LM_Head then
13       WriteLocalDisk(address, data);
14     else
15       Enqueue(address, data);
   // Omit OnTenantRead() and OnRecvAck()
   // In destination SmartNIC:
16 Function OnRecvData(address, data):
17   WriteLocalDisk(address, data);
18   SendAck(source IP, address);
```

synchronous. The LM can only proceed forward after receiving both ACKs. For user reads, the SmartNIC reads the local disk since the local disk has the most up-to-date data, with read-after-write support.

- If the address of a user I/O falls into `inflightMigrating` sub-region, if the I/O is a user write, the LM controller buffers the writes in order, until the lock is released from that specific address (`line15`). For a user read, the LM controller forwards it to the local disk if the address specified in the user read request has no buffered write requests that arrive before the read. Otherwise, the read request is buffered in a queue as well until the address is released from the locking window. Once the lock is released, the address is re-marked from `inflightMigrating` to `Migrated`, and the LM controller conducts the queued user I/Os in order, following the above algorithm to handle I/Os falling into `Migrated` sub-region.

The LM controller does not provide acknowledgment to the VM until double writes are both committed. Later reads/writes to the same address need to wait for the double writes to complete. User commands are not stored in the LM controller but in the VM queues. This reduces buffer requirements on the FPGA for in-flight commands.

Algorithm 2: PYROCUMULUS backpressure algorithm between LM pause and VM user latency.

```
   // In LM controller on FPGA SmartNIC:
1  Procedure Backpressure handler():
2      if !FLAG_LM_PAUSE then
3          if MAX_LAT_PERIOD > PAUSE_TH or net
            bw not enough then
4              pause Live Migration();
5              Time_pause ← Time_cur;
6              FLAG_LM_PAUSE ← 1 ;
7              LM_Head ← LM_Tail;
8      else
9          if Time_cur - Time_pause > PAUSE_DURATION
            then
10             restart Live Migration();
11             FLAG_LM_PAUSE ← 0;
```

6 Live Migration aware Network Management

Given an FPGA SmartNIC-enabled architecture and an FPGA SmartNIC-managed LM algorithm, the remaining challenge is how to manage and share the network bandwidth.

Baseline: Window-based congestion control. We begin with analyzing the FPGA SmartNIC-based design described in Sec. 5. The LM algorithm employs a hardware-assisted, SmartNIC-based double write mechanism that manages a sliding locking window over the LBA region to block in-flight data blocks. This window functions as an implicit congestion control mechanism: by proactively adjusting the window size, the sender effectively throttles transmission and adapts LM copy traffic to use available bandwidth efficiently, as shown in Fig. 6(a). However, this paper shows that window-based congestion control in LM has the following problems.

- This locking window only controls LM copy traffic, without direct control of LM consistency traffic.
- The window-based approach is too implicit, with only indirect bandwidth usage control and without prioritization or QoS support, given that multiple classes of traffic co-exist.
- This mechanism delays user writes from VMs before the locking window's in-flight requests give ACKs back.

PYROCUMULUS relies on the window-based mechanism on the FPGA SmartNIC for hardware simplicity, while providing the following designs to overcome the above drawbacks, stabilizing low user latency and reducing network contention.

Backpressure-enabled network management. We find that when LM copy runs at a high rate, there are latency spikes when measuring VM's local read/write. The root cause of high user latency in the window-based SmartNIC-driven PYROCUMULUS protocol is the absence of a backpressure mechanism[4] between (1) how much LM-related network traffic is needed , and (2) how a VM's local SSD access latency varies.

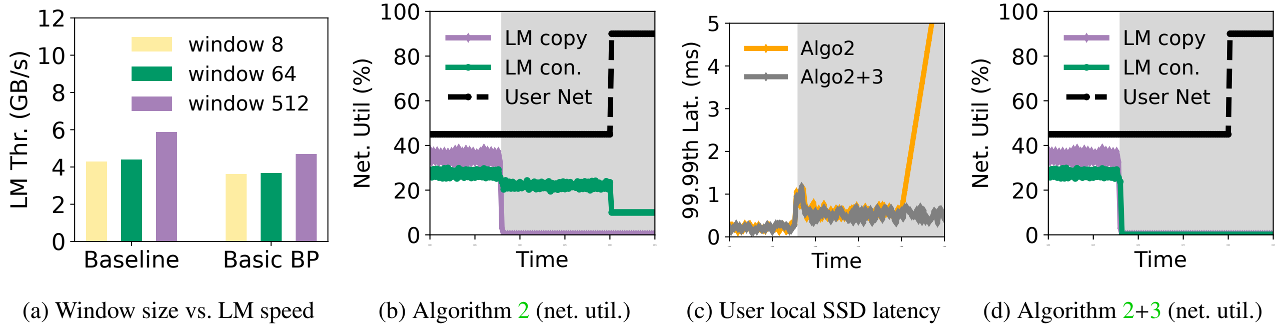
We pause LM when user latency spikes too high. This is theoretically feasible because when pausing LM, we stop blocking requests that fall into `inflightMIGRATING` subregion, which mitigates the queueing of requests. This idea borrows the philosophy of Priority-based Flow Control (PFC), throttling the LM traffic to prioritize user experience.

The major challenge is to make our backpressure effective and hardware-friendly. The LM controller decides (1) when to pause LM, and (2) when to resume LM. Alg. 2 describes how the backpressure feedback is established between the VM user experience on local SSDs and the LM. The provider pre-configures two parameters into the LM controller hardware: (1) the latency threshold to trigger LM pause, $PAUSE_TH$, and (2) the pause duration before resuming, $PAUSE_DURATION$. PYROCUMULUS's pause condition checking and metric tracking has low hardware overhead. First, we only monitor the maximum user latency MAX_LAT_PERIOD within a period (e.g., 100 μs). We trigger a pause when MAX_LAT_PERIOD exceeds $PAUSE_TH$ or when network bandwidth is occupied by the user or core network and unavailable for LM (`line3` in Alg. 2). Second, we use a hardware counter to resume LM after time reaching $PAUSE_DURATION$ (`line9`).

At the beginning of the LM pause, we reset `LM_head` to be `LM_tail` (`line7`). This discards in-flight copies and pauses the LM process immediately. This is crucial to avoid network contention that prevents the LM from pausing, which makes user latency even worse. When resuming the LM, we replay those discarded in-flight copies.

Delaying and rate limiting remote writes: worst case analysis. We observe that Alg. 2 works in most scenarios with proper $PAUSE_TH$ and $PAUSE_DURATION$ set. However, the basic design still suffers from LM-induced network contention even when LM is paused. This is because Alg. 2 only pauses LM copy traffic, while LM consistency traffic (i.e., double writes) still uses the network when user writes fall into `MIGRATED` region. This traffic always exists due to the nature of the double write algorithm. As shown in Fig. 6(b), only LM copy traffic is stopped during the pause time, while LM consistency traffic still takes a non-trivial amount of network bandwidth. We generate user TX traffic, sending 1MB messages using 45% of the network bandwidth; During the pause period, we increase user network traffic up to 90% to study the scenario of network contention during LM. We prioritize user network traffic over LM consistency traffic; LM copy has the lowest priority. We find that when user network traffic dominates, double writes are delayed, and user latency to read and write local SSDs is affected, as shown in Fig. 6(c).

[4] This paper studies the backpressure between VM's local storage latency and the LM, different from L2/L3 backpressure between two nodes.



(a) Window size vs. LM speed (b) Algorithm 2 (net. util.) (c) User local SSD latency (d) Algorithm 2+3 (net. util.)

Figure 6: Network contention studies between LM-related traffic and user latency. The shaded area is the pause period.

Table 3: System setups and workloads of each configuration.

	Hardware	Workload tested
Host Controlled	CPU	FIO, RocksDB, MICA
SoC Offload	CPU + BlueField-2	FIO
Disk Assisted	CPU + Special SSD	FIO
PYROCUMULUS	CPU + Arria 10 FPGA	FIO, RocksDB, MICA

PYROCUMULUS supports full pause, as depicted in Fig. 6(d), where all LM-related traffic is paused. We delay remote writes needed during the pause until the LM is resumed. When a user writes fall into `MIGRATED` region during the pause period, local writes are immediately executed and delivered to the user, and remote writes are kept in a `DelayedWriteTable` on the FPGA SmartNIC. Alg. 3 shows how we augment `SendDoubleWrites` with table-related operations. When the LM is resumed, the function `DelayedTableFlush()` is executed. Since delayed writes are asynchronously independent from users' SSD accesses, we prioritize normal double writes over the delayed writes in the network. In addition, to preclude the bursty traffic of delayed writes from overwhelming the network, we rate limit delayed write traffic at a rate of *DELAYED_WRITE_RATE* (`line10`).

We put the most recently accessed addresses in `DelayedWriteTable` on a 64KB cache located on the FPGA SmartNIC. Each entry of the cache stores a valid bit, a dirty bit, and 8B address. The entire `DelayedWriteTable` is stored in FPGA DRAM.

The hybrid LM algorithm we propose, relying on double write when LM is not paused, while supporting both double writes and dirty page tracking when LM is paused, will be discussed in Sec. 7.6.

Algorithm 3: Rate-limited delayed remote writes.

```
   // Producer of DelayedWriteTable
1  Function SendDoubleWrites(.., address, ..):
2    if !FLAG_LM_PAUSE then
       // Normal double writes...
3    else
4      Insert addr to DelayedWriteTable entry;
5      Enable valid_bit & dirty_bit;
   // Consumer of DelayedWriteTable
6  Function DelayedTableFlush():
7    while DelayedWriteTable not empty do
8      addr ← ReadDelayTable(*entry);
9      data ← ReadLocalDisk(addr);
10     RatedLimitedSendCopyData(..., addr, data,
         FLAG_DWR, DELAYED_WRITE_RATE);
11     Disable valid_bit; *entry ← *entry + 1;
```

7 Experiment Results

7.1 Methodology

This work runs experiments on two Dell R730 servers with dual 16-core Intel Xeon E5 2698v3 CPUs and 256 GB RAM. Ubuntu 18.04 with Linux kernel version 4.17.12 is installed.

Storage-optimized VMs. The VM is created by the QEMU/KVM hypervisor. It is directly mapped to local NVMe SSDs through an internally built SR-IOV interface on the FPGA. The virtual cores are pinned to a group of physical cores for more stable performance.

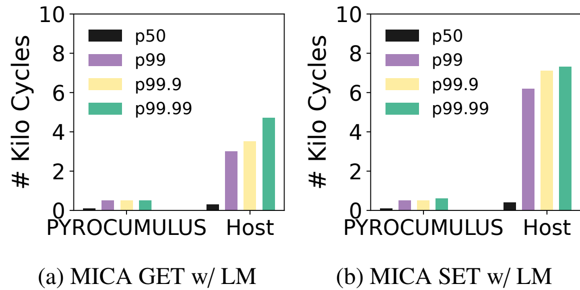
SSDs. Each server has up to 4 PCIe Gen3.0 x 4 NVMe SSDs. Each SSD in use is Intel Optane 900P 280 GB [2]. Local SSDs are mapped to VMs with near-native storage performance.

Networks. Each server has one Mellanox ConnectX-5 NIC [17], which is used by our baselines to perform software-based LM. On the other hand, PYROCUMULUS leverages a 100 Gbps network port on the FPGA SmartNIC for network transfers to/from a remote FPGA SmartNIC.

Comparison configurations. This work tests the system setups listed in Table 3.

Host Controlled: We run a `libvirt` QEMU/KVM hypervisor with local SSD migration, similar to [54, 62]. 5–15 Xeon cores are reserved for LM, with a double write algorithm.

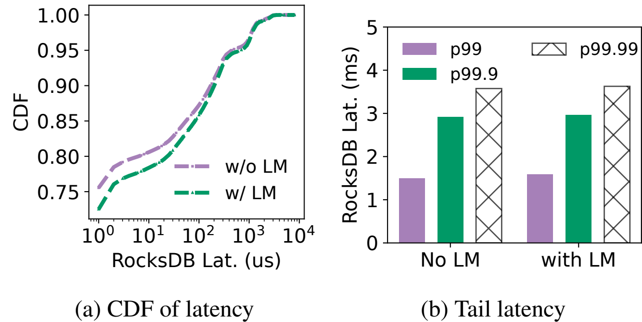
SoC Offload: We use NVidia BlueField SoC [16], where



(a) MICA GET w/ LM

(b) MICA SET w/ LM

Figure 7: Latency of MICA running w/ LM, Pyrocumulus LM vs. Host Controlled LM.



(a) CDF of latency

(b) Tail latency

Figure 8: RocksDB latency on Pyrocumulus w/ and w/o LM.

I/O handling is offloaded to the embedded ARM core on the SoC. We statically provision 7 out of 8 ARM cores for LM, with software based double writes.

Disk Assisted: We emulate its LM algorithm, hardware-assisted dirty page tracking, on Pyrocumulus setting. Each bit tracks the dirtiness of a 1GB region. This emulation is used to measure LM time, which delivers the opportunistic result, because our emulation does not count resource overhead and contention on the CPU. Due to hardware inaccessibility, we cannot accurately evaluate latency variance and resource cost.

User workloads. We measure the SLA, i.e., 99.99th percentile latency, noted as p99.99. We compare users' p99.99 from the start to the end of the LM. First, we use FIO [29] to perform sensitivity studies and explorations for all configurations. Second, we also run `db_bench` [35] on top of the SPDK-based RocksDB, with an ext4 file system installed. Pyrocumulus experiments run from 2 to 8 threads for the RocksDB in the VM. Third, we also run a co-located VM with a latency-sensitive application alongside the VM being migrated. This aims to test the host resource contention on shared caches and host memory bandwidth. We use MICA [51], a memory-intensive key-value store that does not have any SSD accesses.

7.2 Pyrocumulus Implementation

We use an Intel Arria 10 FPGA board with a PCIe Gen 3.0 x8 interface. The FPGA connects to the host and NVMe SSDs through the root complex via PCIe. All the hardware modules are written in Verilog and consist of ~16,000 lines in total.

Network stack. Inter-FPGA communications in the same cluster use Lightweight Transport Layer (LTL) reliable transport protocol [32] with a priority-based flow control (PFC) mechanism. The end-to-end round-trip time between two FPGAs is 4μs, measured by ping messages.

NVMe Stack. The FPGA SmartNIC and NVMe SSD controllers communicate through the PCIe TLP message module. If the remote write is needed, the NVMe stack sends the result to the network stack to perform the LTL connection. Otherwise, the stack initiates the DMA engine to copy the result to the host DRAM. Commands are stored in a 512KB data buffer, which is managed by credit-based buffer allocation. The prototype also provides a register interface for the control plane to configure or access registers of the NVMe stack.

Host stack. Host-SmartNIC communications go through the FPGA shell built-in host DMA stack. It decodes a PCIe transaction and sends I/O commands to the right hardware module.

Driver. This prototype implements the control path functionality for configuring NVMe SSDs in user-space libraries, based on the open source OFA driver [18]. There is an additional software piece with ~200 lines of C code.

Integrating Pyrocumulus LM into host software. The LM controller in Pyrocumulus can be ported to the hypervisor. For example, to work with a QEMU/KVM emulated NVMe interface, this prototype modifies `qemuMonitorBlockdevMirror` in `libvirt`, instead of its original LM implementation in `QEMU_block_job`.

7.3 User Performance Transparency

Live migration co-located with MICA. We inject MICA requests with 50%/50% GET/SET, and it assigns 28 threads on both the source and the destination hosts, one thread per core. For the baseline, we use the LM feature of the QEMU/KVM hypervisor for VMs with local storage. Five cores are statically reserved for Host Controlled LM. Hyperthreading is off. Ideally, LM should not interfere with MICA.

Fig. 7 shows the latency comparisons of MICA SETs and GETs. Pyrocumulus lowers the latency of GETs by 6× and 9.4×, in terms of p99 and p99.99 latencies. Similarly, p99 and p99.99 latencies of MICA SETs decrease due to Pyrocumulus design by 12.4× and 12.1×. Host memory bandwidth contention is the reason for the significant MICA latency increase in the baseline. We measure that 120 MOps MICA throughput would roughly consume 33 GB/s memory bandwidth, and the peak MICA throughput reaches up to 190 MOps, taking up to 86% host memory bandwidth.

Live migration for a VM running RocksDB. This test uses the RocksDB official benchmarking tool `db_bench` [35]

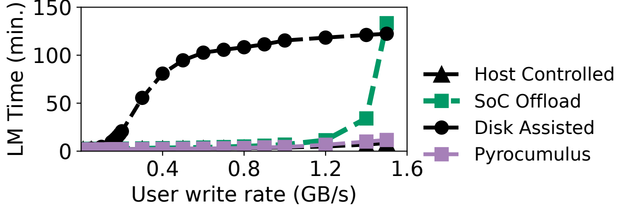


Figure 9: LM time comparison of a 200 GB VM image.

to generate test traffic. RocksDB key size is 30 bytes, and the value size is 30KB. The `readwhilewriting` pattern is measured. Compression is disabled to achieve high SSD bandwidth and exhaustively compete for resources with LM. RocksDB I/Os are prioritized over the LM I/Os by our hardware. This RocksDB workload takes ~7 seconds to initialize; therefore, LM is started at the 10th second to ensure RocksDB is ready when LM starts. We end the RocksDB workload when the LM completes successfully.

We find the Host Controlled LM induces a 9.8% p99.99 latency increase. In contrast, Fig. 8 shows PYROCUMULUS only increases p99.99 latency by 1.4%.

Live migration for a VM running FIO. We run FIO to test the upper-bound of user throughput when co-running with LM. The VM runs 4 KB random writes/reads at 2 GB/s on the source node, with read:write ratio of 1:9. Since FIO is a high performance storage access library that can access local SSDs at line rate with little software overhead. We find p99.99 latency has negligible drops on Host Controlled, SoC Offload, and PYROCUMULUS. However, throughput-wise, PYROCUMULUS outperforms the other two baselines significantly. The throughput with or without LM is nearly the same on PYROCUMULUS. The Host Controlled and SoC Offload have throughput drops of 57% and 75%, respectively. This is because the two baselines statically provision compute resources for LM to avoid latency variances. SoC Offload has a more severe drop than Host Controlled because the SoC is nearly fully occupied to make a LM successful, thus has little compute resources left to offload I/O processing on the SoC [49, 56].

Due to hardware inaccessibility, we cannot accurately evaluate user performance transparency when LM is performed based on the Disk Assisted approach. However, we expect significant CPU resource contention due to the need for log processing and extra I/O handling on the CPU.

7.4 Live Migration Time

We collect the LM time for those successful tests where LM can converge. The VM image size in this test is 200 GB. We run FIO with a read:write ratio of 1:1, with 4KB requests. We increase the FIO request rate from 0 up to 3.5GB/s; thus, the maximal write rate we test is 1.75 GB/s.

Fig. 9 shows comparisons of LM times across different user write rates. First, the LM time of the Disk Assisted increases substantially when the user's random write rate is greater than 170 MB/s. This is due to the dirty page tracking algorithm the Disk Assisted approach uses in the SSD hardware [27], where dirty page tracking granularity is too coarse-grained, at 1GB granularity. Note that this experiment does not include the overhead of offloaded writes in the Disk Assisted approach, as we do not have that type of real hardware SSD. The actual LM time is even higher. The Host Controlled and SoC Offload approaches both implement a double-write algorithm within the host software. While the SoC Offload method delivers low and stable LM time with increasing write rates, its performance degrades sharply beyond 1.2 GB/s. This inflection point is caused by resource exhaustion on the SoC, which is overwhelmed by the latency management software overhead. The Host-Controlled approach maintains low latency across all write rates due to sufficient resource provisioning, albeit at a higher cost, which will be discussed in Sec. 7.5. Overall, at 1.4 GB/s user write rate, PYROCUMULUS delivers 19.6× and 17.8× faster LM, compared with SoC Offload and Disk Assisted LMs, respectively.

Table 4: PYROCUMULUS FPGA usage.

Modules	ALMs	BRAM (Mbits)
NVMe Stack	9,695 (2.3%)	4.296 (7.7%)
Network Stack	6,395 (1.5%)	1.203 (2.2%)
Host Stack	3,738 (0.9%)	2.089 (2.2%)
LM Controller	3,725 (0.9%)	2.09 (3.8%)

Table 5: Comparing minimal costs of a successful LM.

	Hardware Platform	Min Cost
Host Controlled	Intel Xeon E5-2673 v4	15 × ~$100
SoC Offload	NVidia BlueField-2	2 × ~$1,000
PYROCUMULUS	Intel Arria 10 FPGA	0.9% of ~$500

7.5 Cost

Hardware overhead of PYROCUMULUS. The LM controller module is highly resource-efficient. Implementation on an Intel Arria 10 FPGA–a low-cost platform released in 2013–consumes only 0.9% of adaptive logic modules (ALMs) and 3.8% of block RAM (BRAM). We anticipate this overhead would be substantially lower on contemporary SmartNICs such as Azure Boost [19]. A breakdown of hardware overheads in our FPGA SmartNIC prototype is shown in Table 4.

Total cost comparison. We compare the minimal compute resource budgets required for successful LM under random writes at 2 GB/s. As summarized in Table 5, the Host Controlled approach requires 15 CPU cores on an Intel server, with an estimated per-core cost of ~$100. Our analysis shows that a single NVIDIA BlueField-2 SmartNIC (~$1,000) is insufficient for VM migration at this write rate. We estimate that

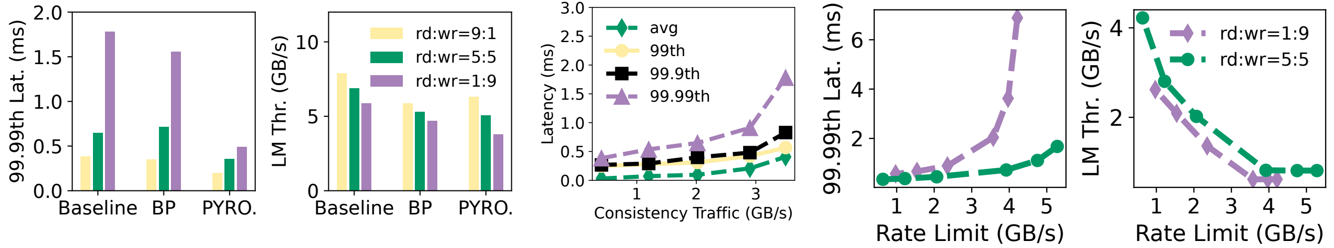
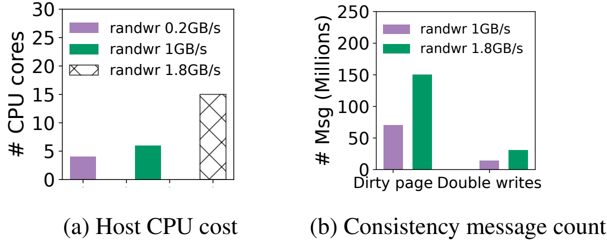


Figure 10: The impact of write intensity. BP: backpressure. PYRO: Pyrocumulus.

Figure 11: Consistency traffic vs. user latency.

Figure 12: The effectiveness of rate limiting.



(a) Host CPU cost

(b) Consistency message count

Figure 13: Study of resource cost on Host Controlled LM.

successful LM requires resources on at least two BlueField-2 cards.[5] In contrast, Pyrocumulus leverages an FPGA (~$500). This achieves estimated cost savings of at least 3–4×, since Pyrocumulus does not dedicate the entire FPGA for LM. Disk Assisted has two inherent cost factors: the substantial CPU resources it still requires and the premium for advanced NVMe SSDs that support SSD-based LM. As the market price for such specialized SSDs is unclear, a direct cost comparison is not feasible.

Root cause of the cost. We attribute high LM cost primarily to substantial software overheads, after studying more on the Host Controlled approach. First, during LM, the host CPU should intercept the user's virtual I/Os, send data and receive acknowledgment over the network, and manage migration I/Os. Such an interception costs up to 80% CPU cycles due to guest interrupts, context switches, and cache pollution [28, 44, 45]. Second, each consistency message triggered by the LM algorithm has high CPU overhead. Each message involves lock synchronization, order checking, and network operations.Worse, the number of consistency messages can grow significantly with the rate of local SSD writes, as shown in Fig. 13(a). Additionally, the double write algorithm generates far fewer consistency messages than dirty page tracking during LM; however, it still produces up to 30 million messages when the user issues random writes at 1.8 GB/s, shown in Fig. 13(b). Since Pyrocumulus customizes an efficient hardware module to handle copying and consistency, the software overheads are eliminated.

[5] Coordinating multiple SoCs for LM is beyond the scope of this paper.

7.6 Sensitivity Studies of Pyrocumulus

User write ratio. Increased write rates result in reduced LM throughput and greater variance in user latency. As depicted in Fig. 10 for a 2GB/s user request rate, Pyrocumulus significantly outperforms alternative approaches. Specifically, under a 90% write ratio, the baseline window-based congestion control and the backpressure algorithm (Alg. 2) exhibit p99.99 latencies 3.6× and 3.2× higher, respectively. For other write ratios (10% and 50%), Pyrocumulus reduces p99.99 latency by factors ranging from 1.78 to 1.94. This performance improvement entails a trade-off: Pyrocumulus sacrifices 20–35% of LM throughput compared to the baseline to maintain low user tail latency. We deem this trade-off acceptable, as the backpressure algorithm also impairs LM progress by employing pauses, yet these pauses are not effective at mitigating user latency variance.

Rate limiting consistency-related traffic. We evaluate Alg. 1 under a user workload running at 2 GB/s on a local Intel Optane 900P SSD, with read:write ratios of 1:9, 3:7, 5:5, and 9:1. Fig. 11 shows that user latency suffers an exponential increase as the volume of consistency traffic grows.

To mitigate this, Pyrocumulus delays remote writes and transmits them asynchronously after resuming a paused LM process (Alg. 1+2+3). As shown in Fig. 12, rate-limiting this delayed traffic to below 2 GB/s effectively reduces p99.99 tail latency and increases LM throughput. This strategy is crucial for LM success under network contention; performing delayed writes lazily after a resume reduces the burst of consistency traffic, thereby alleviating contention. We observe that approximately 10% of delayed remote writes are obsolete, as the target addresses were overwritten with new data before the delayed write could be executed. In that case, we make that address invalid in the `DelayWriteTable` and cancel that particular obsolete write. Fig. 14 shows p99.99 latency variance over time. Pyrocumulus's full design, combining our three algorithms, delivers much lower user latency.

Hotness of address accesses. The double writes algorithm we use has the downside that VM's I/O performance is directly coupled to the network and target device performance; an I/O operation cannot be acknowledged to the VM until it is mirrored. Thus, workloads with high write locality to a few blocks can generate significant network traffic. In practice,

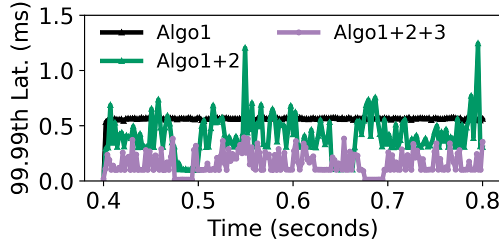


Figure 14: Effectiveness of network contention avoidance.

the guest OS cache can mitigate this scenario. In addition, Pyrocumulus' delayed remote write during pause is a dirty page tracking like algorithm, which can balance the downside of double writes. Thus, Pyrocumulus **hybridizes two algorithms**, making VM and LM more stable and robust. For example, we can explicitly control the migration convergence by deciding when to resume LM.

FPGA mediation performance overhead. During LM, all I/Os from the VM have to go through the FPGA SmartNIC, adding an extra PCIe hop between the CPU and local SSDs. Latency-wise, 1–2μs added by such mediation is negligible compared with the long tail latency of disk I/Os. Throughput-wise, the FPGA will not become the bottleneck since it has enough parallelism to handle intensive I/Os traffic up to tens of GB/s. When LM does not happen, VM accesses do not have to go through the FPGA SmartNIC; they can still go through the VM–SSD native path.

8 Related Work and Discussions

SmartNICs in the cloud. Leveraging SmartNICs to offload critical features such as LM is practical and deployable. Production-ready solutions using SmartNIC-enabled systems have emerged in current cloud platforms, such as AWS Nitro [9], Azure AccelNet [36], and Alibaba Cloud CIPU [7]. Therefore, there are no extra capital expenditures when deploying Pyrocumulus into clouds that have FPGA SmartNICs. In addition, cloud providers go to great lengths to offload suitable applications onto SmartNICs and minimize costs on the host [34, 37, 40, 41, 46–48, 52, 53, 57, 61, 63–65]. Its applicability ranges from network, storage, to virtualization [32, 36, 45, 56]. Pyrocumulus is the first to offload LM on SmartNICs and the first to propose hardware-based LM.

Live migration. The most related proposals are LM techniques for VMs with local SSDs [54, 62]. However, they do not consider workload characteristics for storage-optimized VMs. Pyrocumulus also distinguishes from the two works in terms of the SmartNIC offload, hardware customization, and enabling LM pause/resume, rather than pausing VM or cancelling LM [54, 62]. Additionally, Pyrocumulus relies on double write when LM is not paused, while supporting both double writes and dirty page tracking when LM is paused. Finally, Pyrocumulus targets local storage, while other proposals have other LM resource targets, ranging from memory [39, 43, 62], RDMA connections [50, 59] and network states [33, 58, 60, 62, 66, 67].

Why FPGA SmartNICs are better than SoC SmartNICs for LM. The high resource cost of LM makes it impossible for many LM jobs on SoC SmartNICs to finish; these jobs easily fail because we do not have enough SoC resources to provision for LM. Worse, the LM resource provisioning takes up nearly the entire SoC just for LM, losing SoC resources to support other important offload features such as virtualization and network services.

9 Conclusions

This work enables live migration (LM) for storage-optimized VMs that have directly mapped local SSDs, low tail latency SLA targets, and intensive disk accesses. The **unique challenge of this research problem is that LM needs a minimal compute/memory budget and dynamic network contention management**. Enabling LM for storage-optimized VMs comes with high resource overheads. Providers cannot hope to avoid high resource usage by simply allocating fewer resources and accepting a slower LM time – **LM is a win-or-lose game**. In fact, LM cannot converge successfully if LM resources are not sufficient; LM will fail even if LM resources are enough during most of the LM time. Therefore, the undesirable consequence is that the inevitable high overhead leads to high costs because non-trivial LM compute/memory resources must be statically provisioned for a successful LM. Worse, LM is also vulnerable to network contention.

Pyrocumulus offers user-transparent, fast, and low-cost LM on FPGA SmartNICs. Contributions are threefold. First, we are the first to target LM storage-optimized VMs. We show that LM compute/memory resources are feasible to be customized in low-overhead hardware, therefore avoiding high resource costs induced by software overhead. Second, we propose to statically provision a small hardware module on FPGA SmartNICs. We also leverage SmartNIC's efficient network access and provide LM-aware network contention. Unlike prior work, we enable LM pause, with a hybrid LM algorithm to control LM traffic overhead, paused or unpaused. Third, Pyrocumulus lowers the bar of deployment without adding extra capital expenditures, introducing resource contention, or user performance interference.

Acknowledgment

We would like to thank the shepherd Boris Pismenny and our anonymous reviewers for their valuable feedback. We thank Microsoft Research Asia's Star of Tomorrow Program. This work was supported in part by a Canada Research Chair.

References

[1] Intel arria 10 gx 160 - maximum resources. https://www.intel.com/content/www/us/en/docs/programmable/683332/current/maximum-resources-34722.html. Accessed: 2025-7-2.

[2] Intel® optane ssd 900p series 280gb 12 height pcie x4 20nm 3d xpoint product specifications. https://ark.intel.com/content/www/us/en/ark/products/123628/intel-optane-ssd-900p-series-280gb-12-height-pcie-x4-20nm-3d-xpoint.html. Accessed: 2024-7-8.

[3] Kernel-based virtual machine (kvm). http://www.linux-kvm.org. Accessed: 2024-7-8.

[4] Microsoft hyper-v. http://www.microsoft.com/en-us/server-cloud/solutions/virtualization.aspx. Accessed: 2024-7-8.

[5] Vmware. http://www.vmware.com. Accessed: 2024-7-8.

[6] Broadcom stingray ps1100r. https://docs.broadcom.com/doc/PS1100R-PB, 2022. Accessed: 2022-1-1.

[7] Alibaba cipu. https://www.breakinglatest.news/business/alibaba-clouds-first-cipu-processor-is-reverse-self-developed-for-os/, 2024. Accessed: 2024-7-8.

[8] Amazon dynamodb. https://aws.amazon.com/dynamodb/, 2024. Accessed: 2024-7-8.

[9] Aws nitro system. https://aws.amazon.com/ec2/nitro/, 2024. Accessed: 2024-7-8.

[10] Azure disk storage overview - azure virtual machines | microsoft learn. https://learn.microsoft.com/en-us/azure/virtual-machines/managed-disks-overview, 2024. Accessed: 2024-7-8.

[11] Azure metadata service: Scheduled events for windows vms. https://learn.microsoft.com/en-us/azure/virtual-machines/windows/scheduled-events, 2024. Accessed: 2024-7-8.

[12] Cloud block storage - amazon ebs - aws. https://aws.amazon.com/ebs/, 2024. Accessed: 2024-7-8.

[13] Google cloud - live migration process during maintenance events. https://cloud.google.com/compute/docs/instances/live-migration-process, 2024. Accessed: 2024-7-8.

[14] Maintenance and updates - azure virtual machines. https://learn.microsoft.com/en-us/azure/virtual-machines/maintenance-and-updates#live-migration, 2024. Accessed: 2024-7-8.

[15] Nvidia® bluefield® snap hardware-accelerated virtualization of nvme storage. https://docs.nvidia.com/networking/display/bluefielddpuosv470/bluefield+snap, 2024. Accessed: 2024-7-8.

[16] Nvidia® mellanox® bluefield-2 smartnic. https://www.mellanox.com/products/bluefield2-overview, 2024. Accessed: 2024-7-8.

[17] Nvidia® mellanox® connectx-5. https://www.nvidia.com/en-us/networking/ethernet/connectx-5/, 2024. Accessed: 2024-7-8.

[18] Open fabrics nvme driver. https://nvmexpress.org/open-fabrics-alliance-nvm-express-window-driver-1-4-released-december-8-2014/, 2024. Accessed: 2024-7-8.

[19] Overview of azure boost. https://learn.microsoft.com/en-us/azure/azure-boost/overview, 2024. Accessed: 2024-7-8.

[20] Persistent disk: Durable block storage | google cloud. https://cloud.google.com/persistent-disk, 2024. Accessed: 2024-7-8.

[21] Storage-optimized instances - amazon ec2. https://docs.aws.amazon.com/ec2/latest/instancetypes/so.html, 2024. Accessed: 2024-7-8.

[22] Storage-optimized virtual machine sizes - azure virtual machines. https://learn.microsoft.com/en-us/azure/virtual-machines/sizes-storage, 2024. Accessed: 2024-7-8.

[23] Temporary disk - overview of azure disk storage - azure virtual machines. https://learn.microsoft.com/en-us/azure/virtual-machines/managed-disks-overview#temporary-disk, 2024. Accessed: 2024-7-8.

[24] Aws - live migration process during maintenance events. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/InstanceStorage.html, 2025. Accessed: 2025-5-8.

[25] Qemu: A generic and open source machine emulator and virtualizer. https://www.qemu.org/, 2025. Accessed: 2025-7-8.

[26] Mike Allison. NVM Express (NVMe) Host Controlled Data Center Migration for Hyperscalers and the Enterprise. https://nvmexpress.org/nvm-express-nvme-host-controlled-data-center-migration-for-hyperscalers-and-the-enterprise/, 2024. Accessed: 2024-9-1.

[27] Mike Allison. The NVM Express(R) ratified details of live migration. https://www.sniadeveloper.org/events/agenda/session/731, 2024. Accessed: 2024-9-1.

[28] Nadav Amit, Muli Ben-Yehuda, Dan Tsafrir, and Assaf Schuster. viommu: Efficient iommu emulation. In *Proceedings of the 2011 USENIX Annual Technical Conference (ATC)*, pages 73–88, 2011.

[29] Jens Axboe. Flexible i/o tester. https://github.com/axboe/fio. Accessed: 2024-7-8.

[30] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, pages 164–177, 2003.

[31] Catherine Burke. Improving azure virtual machine resiliency with predictive ml and live migration. https://azure.microsoft.com/en-us/blog/improving-azure-virtual-machine-resiliency-with-predictive-ml-and-live-migration/?msockid=3ddf1046d62469d31f7e0483d75e6813, 2018. Accessed: 2024-7-8.

[32] Adrian M. Caulfield, Eric S Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. A cloud-scale acceleration architecture. In *Proceedings of the 49th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13, 2016.

[33] Inho Choi, Nimish Wadekar, Raj Joshi, Joshua Fried, Dan R.K. Ports, Irene Zhang, and Jialin Li. Capybara: μsecond-scale live tcp migration. In *Proceedings of the 14th ACM Asia-Pacific Workshop on Systems (APSys)*, pages 30–36, 2023.

[34] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Maleen Abeydeera, Logan Adams, Hari Angepat, Christian Boehn, Derek Chiou, Oren Firestein, Alessandro Forin, Kang Su Gatlin, Mahdi Ghandi, Stephen Heil, Kyle Holohan, Ahmad El Husseini, Tamas Juhasz, Kara Kagi, Ratna K. Kovvuri, Sitaram Lanka, Friedel van Megen, Dima Mukhortov, Prerak Patel, Brandon Perez, Amanda Grace Rapsang, Steven K. Reinhardt, Bita Darvish Rouhani, Adam Sapek, Raja Seera, Sangeetha Shekar, Balaji Sridharan, Gabriel Weisz, Lisa Woods, Phillip Yi Xiao, Dan Zhang, Ritchie Zhao, and Doug Burger. Serving dnns in real time at datacenter scale with project brainwave. *IEEE Micro*, 38(2):8–20, 2018.

[35] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. Rocksdb: evolution of development priorities in a key-value store serving large-scale applications. *ACM Transactions on Storage (TOS)*, 17(4):1–32, 2021.

[36] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. Azure accelerated networking: Smartnics in the public cloud. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 51–66, 2018.

[37] Stewart Grant, Anil Yelam, Maxwell Bland, and Alex C. Snoeren. Smartnic performance isolation with fairnic: Programmable networking for the cloud. In *Proceedings of the 2020 ACM SIGCOMM Conference*, pages 681–693, 2020.

[38] Dan Helmick. Live migration for PCIe SSDs by Samsung. https://www.sniadeveloper.org/events/agenda/session/522, 2023. Accessed: 2024-9-1.

[39] Michael R Hines and Kartik Gopalan. Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. In *Proceedings of the 5th ACM international conference on Virtual execution environments (VEE)*, pages 51–60, 2009.

[40] Stephen Ibanez, Muhammad Shahbaz, and Nick McKeown. The case for a network fast path to the cpu. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks (HotNets)*, pages 52–59, 2019.

[41] Antoine Kaufmann, Simon Peter, Thomas Anderson, and Arvind Krishnamurthy. Flexnic: Rethinking network dma. In *Proceedings of the 15th ACM Workshop on Hot Topics in Operating Systems (HotOS)*, 2015.

[42] Jongyul Kim, Insu Jang, Waleed Reda, Jaeseong Im, Marco Canini, Dejan Kostić, Youngjin Kwon, Simon Peter, and Emmett Witchel. Linefs: Efficient smartnic offload of a distributed file system with pipeline parallelism. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP)*, pages 756–771, 2021.

[43] Chinmay Kulkarni, Aniraj Kesavan, Tian Zhang, Robert Ricci, and Ryan Stutsman. Rocksteady: Fast migration for low-latency in-memory storage. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, pages 390–405, 2017.

[44] Yossi Kuperman, Eyal Moscovici, Joel Nider, Razya Ladelsky, Abel Gordon, and Dan Tsafrir. Paravirtual remote i/o. In *Proceedings of the 21st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 49–65, 2016.

[45] Dongup Kwon, Junehyuk Boo, Dongryeong Kim, and Jangwoo Kim. Fvm: Fpga-assisted virtual device emulation for fast, scalable, and flexible storage virtualization. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 955–971, 2020.

[46] Yanfang Le, Hyunseok Chang, Sarit Mukherjee, Limin Wang, Aditya Akella, Michael M. Swift, and T.V. Lakshman. Uno: Uniflying host and smart nic offload for flexible packet processing. In *Proceedings of the 8th ACM Symposium on Cloud Computing (SoCC)*, pages 506–519, 2017.

[47] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. Kv-direct: High-performance in-memory key-value store with programmable nic. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, pages 137–152, 2017.

[48] Bojie Li, Kun Tan, Layong Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. Clicknp: Highly flexible and high performance network processing with reconfigurable hardware. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 1–14, 2016.

[49] Huaicheng Li, Mingzhe Hao, Stanko Novakovic, Vaibhav Gogte, Sriram Govindan, Dan R.K. Ports, Irene Zhang, Ricardo Bianchini, Haryadi S. Gunawi, and Anirudh Badam. Leapio: Efficient and portable virtual nvme storage on arm socs. In *Proceedings of the 25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 591–605, 2020.

[50] Xiaoyu Li, Ran Shu, Yongqiang Xiong, and Fengyuan Ren. Software-based live migration for rdma. In *Proceedings of the 2025 ACM SIGCOMM Conference*, pages 99–113, 2025.

[51] Hyeontaek Lim, Dongsu Han, David Andersen, and Michael Kaminsky. Mica: A holistic approach to fast in-memory key-value storage. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 429–444, 2014.

[52] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading distributed applications onto smartnics using ipipe. In *Proceedings of the 2019 ACM SIGCOMM Conference*, pages 318–333. 2019.

[53] Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. E3: energy-efficient microservices on smartnic-accelerated servers. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, pages 363–378, 2019.

[54] Ali José Mashtizadeh, Emré Celebi, Tal Garfinkel, and Min Cai. The design and evolution of live storage migration in vmware esx. In *Proceedings of the 2011 USENIX Annual Technical Conference (ATC)*, pages 187–200, 2011.

[55] Sasha Melamed. New storage-optimized azure vms deliver higher performance for data intensive workloads. https://techcommunity.microsoft.com/blog/azurecompute/new-storage-optimized-azure-vms-deliver-higher-performance-for-data-intensive-wo/3403356, 2022. Accessed: 2024-11-8.

[56] Jaehong Min, Ming Liu, Tapan Chugh, Chenxingyu Zhao, Andrew Wei, In Hwan Doh, and Arvind Krishnamurthy. Gimbal: enabling multi-tenant storage disaggregation on smartnic jbofs. In *Proceedings of the 2021 ACM SIGCOMM Conference*, pages 106–122, 2021.

[57] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. Floem: a programming system for nic-accelerated network applications. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 663–679, 2018.

[58] Maksym Planeta, Jan Bierbaum, Leo Sahaya Daphne Antony, Torsten Hoefler, and Hermann Härtig. Migros: Transparent live-migration support for containerised rdma applications. In *Proceedings of the 2021 USENIX Annual Technical Conference (ATC)*, pages 47–63, 2021.

[59] Artem Y. Polyakov, Gal Shalom, Asaf Schwartz, Aviad Yehezkel, Omri Ben David, Omri Kahalon, Ariel Shahar, and Liran Liss. Device-assisted live migration of rdma devices. In *Proceedings of the 31st ACM Symposium on Operating Systems Principles (SOSP)*, pages 153–168, 2025.

[60] George Prekas, Marios Kogias, and Edouard Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, pages 325–341, 2017.

[61] Sivasankar Radhakrishnan, Yilong Geng, Vimalkumar Jeyakumar, Abdul Kabbani, George Porter, and Amin Vahdat. Senic: Scalable nic for end-host rate limiting. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 475–488, 2014.

[62] Adam Ruprecht, Danny Jones, Dmitry Shiraev, Greg Harmon, Maya Spivak, Michael Krebs, Miche Baker-Harvey, and Tyler Sanderson. Vm live migration at scale. In *Proceedings of the 14th ACM International Conference on Virtual Execution Environments (VEE)*, pages 45–56, 2018.

[63] Henry N. Schuh, Weihao Liang, Ming Liu, Jacob Nelson, and Arvind Krishnamurthy. Xenic: Smartnic-accelerated distributed transactions. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP)*, pages 740–755, 2021.

[64] Brent Stephens, Aditya Akella, and Michael Swift. Loom: Flexible and efficient nic packet scheduling. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 33–46, 2019.

[65] Tushar Swamy, Alexander Rucker, Muhammad Shahbaz, Ishan Gaur, and Kunle Olukotun. Taurus: a data plane architecture for per-packet ml. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 1099–1114, 2022.

[66] Xin Xu and Bhavesh Davda. Srvm: Hypervisor support for live migration with passthrough sr-iov network devices. In *Proceedings of the 12th ACM International Conference on Virtual Execution Environments (VEE)*, pages 65–77, 2016.

[67] Jiechen Zhao, Iris Uwizeyimana, Karthik Ganesan, Mark C. Jeffrey, and Natalie Enright Jerger. Altocumulus: Scalable scheduling for nanosecond-scale remote procedure calls. In *Proceedings of the 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 423–440, 2022.