TeRM: Extending RDMA-Attached Memory with SSD

Zhe Yang, Qing Wang, Xiaojian Liao, and Youyou Lu, *Tsinghua University;*
Keji Huang, *Huawei Technologies Co., Ltd;* Jiwu Shu, *Tsinghua University*

https://www.usenix.org/conference/fast24/presentation/yang-zhe

This paper is included in the Proceedings of the
22nd USENIX Conference on File and Storage Technologies.

February 27–29, 2024 • Santa Clara, CA, USA

978-1-939133-38-0

Open access to the Proceedings
of the 22nd USENIX Conference on
File and Storage Technologies
is sponsored by

TeRM: Extending RDMA-Attached Memory with SSD

Zhe Yang[†], Qing Wang[†], Xiaojian Liao[†], Youyou Lu[†], Keji Huang[‡], and Jiwu Shu[‡]

[†]Tsinghua University
[‡]Huawei Technologies Co., Ltd

Abstract

RDMA-based in-memory storage systems offer high performance but are restricted by the capacity of physical memory. In this paper, we propose TeRM to extend RDMA-attached memory with SSD. TeRM achieves fast remote access on the SSD-extended memory by eliminating page faults of RDMA NIC and CPU from the critical path. We also introduce a set of techniques to reduce the consumption of CPU and network resources. Evaluation shows that TeRM performs close to the performance of the ideal upper bound where all pages are pinned in the physical memory. Compared with existing approaches TeRM significantly improves the performance of unmodified RDMA-based storage systems, including a file system and a key-value system.

1 Introduction

RDMA networks are catalyzing innovative designs for a wide range of in-memory storage systems, including file systems [12, 25, 41], key-value stores [26, 27, 37], and transactional databases [14, 15, 34, 38]. Unlike traditional TCP/IP networks, RDMA can expose server-side memory regions, i.e., *RDMA-attached memory*, to clients in the form of virtual addresses. Clients can directly access data in these regions via one-sided requests. The execution of one-sided requests at the server side bypasses the CPU: the RDMA NIC (RNIC) performs virtual-to-physical address translation using RNIC page table, and then DMAs data to physical memory. In this way, RDMA provides low latency and high CPU efficiency.

However, memory is an expensive and limited resource in datacenters [23, 39]. To improve cost-efficiency and accommodate larger-than-memory data sets for RDMA-based systems, it is desirable to exploit SSD to extend the space of RDMA-attached memory by performing *demand paging* with the physical memory and the SSD. A hardware mechanism called ODP (On-Demand Paging) MR (memory region) [22] is proposed to support it. When handling an RDMA request with the ODP MR, the RNIC will trigger a page fault interrupt for SSD-resident data, then the CPU promotes data from SSD to memory and updates the RNIC page table.

Unfortunately, our experiments demonstrate that ODP MR is not the silver bullet to extend RDMA-attached memory with SSD. As an RDMA READ consumes only 3.66μs on an in-memory page, the latency grows to 570.74μs on an SSD-resident page. The root cause is that the RNIC hardware has limited compute and memory resources [22], so it can only handle exceptions of RNIC page faults in a simple but *inefficient* manner (e.g., discard the received data and notify the client-side RNIC to retransmit it).

Motivated by the analysis above, we propose TeRM, an efficient approach to extending RDMA-attached memory with SSD. The key idea is to onload exception handling (i.e., RNIC page fault) from hardware to software. For all the SSD-resident pages, TeRM makes the RNIC page table point to a reserved physical page containing a predefined magic pattern. In this way, the RNIC page fault is eliminated. For a read request, the client first fetches data through an RDMA READ and identifies whether the page is on the SSD. Then, the client resorts to RPCs to retrieve an SSD-resident page from the server, but does not require any additional operation for memory-resident pages, ensuring fast remote accesses in common cases. Meanwhile, we introduce a set of techniques to reduce the network traffic.

The TeRM-induced RPCs will access SSD-extended virtual memory. To eliminate the heavy CPU page fault [11, 30, 42] from the critical path of RPC execution, we propose tiering IO. The key idea is to access the SSD-extended virtual memory via file IO interfaces instead of memory `load`/`store` interfaces. It reads/writes the SSD-extended virtual memory via buffer IO when the data is cached in the physical memory, and otherwise via direct IO that bypasses the page cache.

With the design techniques above, TeRM eschews both RNIC and CPU page faults from the critical path. However, it freezes data placement on the server, unfortunately. If a hotspot is on the SSD, it will always be accessed by RPC with direct IO. Therefore, TeRM designs a dynamic hotspot

*Jiwu Shu is the corresponding author (shujw@tsinghua.edu.cn).

promotion mechanism, which relies on collaborative effort from clients and the server.

We implement TeRM by building a userspace library tLib with about 6,100 LoC, and modifying the Mellanox RNIC driver with about 300 LoC. tLib overrides the APIs of `libibverbs` and is compatible with existing RDMA applications. Using a microbenchmark, we demonstrate that TeRM achieves 98.13% throughput of the ideal upper bound with half physical memory. We also evaluate unmodified RDMA-based storage systems, a file system, Octopus [25], and a key-value system, XStore [37]. The results show that TeRM outperforms the ODP MR and the software-only RPC approach by up to 642.23× and 7.68×. We open source TeRM at https://github.com/thustorage/TeRM.

To sum up, we make the following contributions.

- We conduct an in-depth breakdown and analysis of the end-to-end latency to access the ODP MR.
- We propose TeRM, an efficient approach to extending RDMA-attached memory with SSD. It onloads exception handling (i.e., RNIC page fault) from hardware to software. We also introduce a set of techniques to reduce network traffic and CPU overhead.
- We use microbenchmarks and unmodified RDMA-based storage systems to demonstrate the effectiveness of TeRM.

2 Background

2.1 RDMA

RDMA registers and initializes two important resources on the control path, queue pair (QP) and memory region (MR). QP is the communication endpoint with another peer. MR exposes an area in the application's virtual memory, for the RNIC to access. Atop the initialized QP and MR, RDMA supports two types of requests on the data path, one-sided and two-sided. READ and WRITE are typical one-sided requests. RDMA applications leverage them to access data on a remote MR, without involving the remote CPU. SEND and RECV (receive) are typical two-sided requests that offer a message-passing abstraction. They are usually used to build RPC.

MR plays an indispensable role in freeing the remote CPU from being interrupted by a one-sided request. While initializing an MR, the driver pins all the pages in the physical memory, retrieves the virtual-to-physical mappings from the CPU page table, and stores them in the RNIC page table. We call the MR initialized in this way a *pinned MR* in the paper. With the pinned MR, the RNIC finds the physical addresses of the target virtual addresses in a one-sided request and accesses the data directly on the physical memory.

Although the pinned MR is prevalent in RDMA applications, it has several limitations. It pins a large number of pages in the physical memory (e.g., tens or hundreds of GBs), occupying valuable DRAM resources. The application can only initialize an MR no larger than the available physical memory.

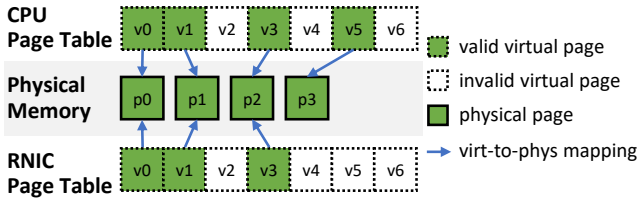

Figure 1: ODP MR. *We show the RNIC page table of an ODP MR and compare it with the CPU page table. A valid virtual page is mapped to a physical page in the page table. An invalid virtual page is not mapped. v5 is valid in the CPU page table but invalid in the RNIC page table. We explain the figure detailedly in §2.2.*

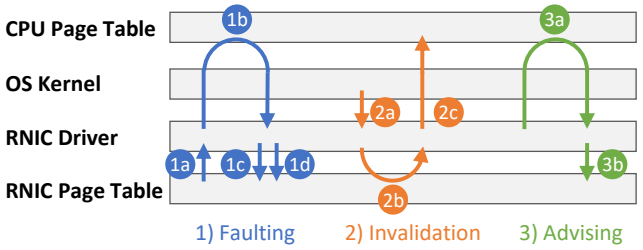

Figure 2: Flows to Synchronize CPU and RNIC Page Tables.

Meanwhile, it loses the opportunities for overcommitment, page migration, transparent huge-page, etc. Facing these limitations, ODP (On-Demand Paging) MR is proposed [22].

2.2 ODP MR

An ODP MR differs from a pinned MR in that it does not pin pages in the physical memory, as we depict in Figure 1. The RNIC page table maps some virtual pages to physical pages — we call them valid virtual pages — and leaves the rest unmapped, i.e., invalid virtual pages. Since the pages are no longer pinned, the OS kernel can swap and migrate pages. The application is able to expose an MR larger than the physical memory. As the virtual-to-physical mappings may be changed, CPU and RNIC page tables are synchronized by three flows illustrated in Figure 2.

1) Faulting. When an RDMA request accesses data on invalid virtual pages, (1a) the RNIC stalls the QP and raises an RNIC page fault[1] interrupt. (1b) The driver requests the OS kernel for virtual-to-physical mappings via `hmm_range_fault` [2]. The OS kernel triggers CPU page faults on these virtual pages and fills the CPU page table if necessary. (1c) The driver updates the mappings on the RNIC page table and (1d) resumes the QP.

2) Invalidation. When the OS kernel tries to unmap virtual

[1] we call the RNIC-triggered page fault an RNIC page fault in this paper, to distinguish it from a CPU page fault triggered by `load`/`store` in this paper

pages in scenarios like swapping out or page migration, (2a) it notifies the RNIC driver to invalidate virtual pages via `mmu_interval_notifier` [2]. (2b) The RNIC driver erases the virtual-to-physical mapping from the RNIC page table. (2c) The driver notifies the kernel that the physical pages are no longer used by the RNIC. Then, the OS kernel modifies the CPU page table and reuses the physical pages.

ODP MR relies on faulting and invalidation flows to synchronize CPU and RNIC page tables. All the valid virtual pages in the RNIC page table are guaranteed valid in the CPU page table, but not vice versa. When the kernel changes an invalid virtual page to a valid one, it does not inform the driver. As we illustrate in Figure 1, v5 is valid in the CPU page table but still left invalid in the RNIC page table.

3) Advising flow tackles the issue above. An application can proactively request the RNIC driver to populate a range in the RNIC page table. The RNIC driver completes advising by steps (3a) – (3b), which are identical to steps (1b) – (1c).

3 Motivation

In this section, we introduce RDMA-attached memory and analyze how ODP MR performs in extending it with SSD. We summarize two principles for designing TeRM.

3.1 RDMA-Attached Memory

An RDMA cluster includes several server and client machines that are equipped with RNICs and connected by RDMA network. By exposing the server's virtual memory with an MR, clients can directly read and write the RDMA-attached memory through RDMA READ/WRITE. Clients and servers may also exchange messages and data through RPC based on RDMA SEND/RECV. The RDMA-attached memory targets storage systems, e.g., file system and key-value system.

Note that the server's virtual memory is accessed both locally and remotely. Local accesses are from the CPU via `load`/`store`. Remote accesses are from clients via RDMA READ/WRITE. We take Octopus [25], an RDMA-based file system, as an example. The Octopus server initializes memory layout, maintains file metadata, and boots an RPC service for receiving and handling metadata requests. After retrieving file metadata (e.g., data addresses) via RPC, the Octopus client directly reads/writes the server-side MR to access file data.

As the server typically registers a pinned MR, it is restricted by the capacity of physical memory, as we explain in §2.1. To improve cost-efficiency and accommodate larger-than-memory data sets, we explore extending the RDMA-attached memory with SSD in this paper.

3.2 ODP MR Is Not the Silver Bullet

ODP MR enables a straightforward approach to extending RDMA-attached memory with SSD. The server-side application `mmap`-s an SSD to get a virtual memory area that exceeds the physical memory. Then it initializes an ODP MR for clients to access remotely.

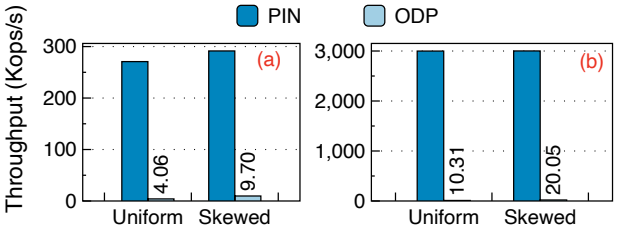

Figure 3: Read Throughput with (a) One Client Thread and (b) 64 Client Threads. *PIN: pinned MR. ODP: ODP MR. Read size: 4KB. MR size: 64GB. Physical memory for ODP MR: 32GB. SSD: Intel Optane P5800X. More detailed experimental setups are listed in §6.1.*

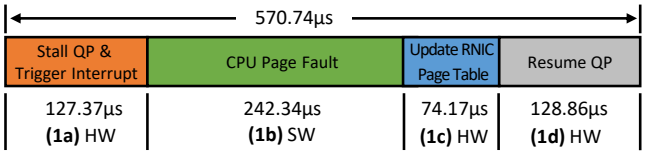

Figure 4: Time Breakdown of Read 4KB on an ODP MR. *(1a) – (1d): the steps introduced in Figure 2. HW: the step is executed by the RNIC hardware. SW: the step is executed by the software on the CPU.*

We use a microbenchmark to evaluate the performance of accessing a server-side ODP MR from the client. We `mmap` 64GB virtual memory from an Intel Optane P5800X SSD, and initialize a pinned MR (annotated as PIN) and an ODP MR (annotated as ODP) respectively; physical memory is limited to 32GB for the ODP MR. We evaluate the throughput of reading 4KB data on the MR. §6.1 offers more details about experimental setups. Figure 3 reports the results. The pinned MR outperforms the ODP MR by 66.64× with one client thread. The gap grows greatly to 290.76× with 64 client threads. The experiment shows that ODP MR exhibits poor performance, which is also reported by other works [16, 36]. Therefore, ODP MR is not the silver bullet for extending RDMA-attached with SSD.

We break down the end-to-end time to read 4KB on an ODP MR that triggers the RNIC page fault. Figure 4 depicts the time of four steps we introduce in §2.2. We do not draw the time of transferring 4KB data after resolving the RNIC page fault, because it occupies less than 5μs, which is negligible in the whole time. Notably, the CPU page fault (step(1b)) in our experiment is a major one where the OS kernel swaps data between the physical memory and the SSD, instead of a minor one [32]. We also evaluate the end-to-end latency with a minor page fault for comparison, which is 431.22μs. The latency difference stems from the software overhead of page

cache mechanisms to access the SSD, as the Intel Optane P5800X SSD has a read/write latency of only about 10μs.

The end-to-end time is composed of hardware time on the RNIC (steps (1a), (1c), and (1d)) and software time on the CPU (step (1b)). As shown in the figure, hardware steps take up more than half of the whole. When identifying an invalid virtual page during processing an RDMA request (step (1a)), the server-side RNIC returns a receiver-not-ready (RNR) negative acknowledgment packet (NACK) to the client-side RNIC [16, 22]. Then the QP is stalled on the request until it is resumed by step (1d). We presume that the latency of steps (1a) and (1d) arises from changing the QP state, which is reported to take up about 100μs [10, 40].

The root cause of the hardware's long latency is its inefficiency in handling exception cases. The limited compute and memory resources of the RNIC result in the simple approach of stalling the transmission. The RNIC circuitry of handling exception path operates relatively slowly, compared to the fast path of processing a normal RDMA request. Therefore, complex handling logic is too difficult to implement, as reported by researchers from Mellanox [22]. Given the above, we propose the first principle for designing TeRM. **Principle #1: onload exception handling from hardware to software.**

The other source of the end-to-end latency is software, the CPU page fault. The CPU page fault is known to perform poorly [11, 21, 28, 29] and does not scale well with the number of threads [30]. However, ODP MR makes the case even worse. During handling a CPU page fault, the kernel recycles a physical page, invalidates the virtual page mapped to it, and finally reuses it for the faulting virtual address. As we describe in §2.2, the kernel triggers the invalidation flow when invalidating a virtual page, where the driver spends considerable time updating the RNIC page table. Thus, the long latency of CPU page fault shown in Figure 4 implicitly includes invalidating the RNIC page table. Considering the above, we propose the second principle for designing TeRM. **Principle #2: eliminate CPU page faults from the critical path.**

4 Design

4.1 Overview

Figure 5 shows the overview of TeRM. We explain it below.

4.1.1 Architecture

Cluster infrastructure. The cluster has several servers and clients; we draw one server and one client in Figure 5 due to space limit. They are equipped with RNICs and connected via RDMA network. tLib is TeRM's userspace library (§5). It has two instances on the server (tLib-S) and the client (tLib-C). **CPU VM** serves local access, i.e., CPU `load`/`store` from the server-side application. The server creates an area of virtual memory larger than the physical memory through `mmap`-ing

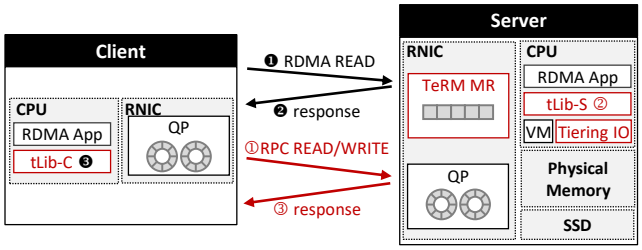

Figure 5: TeRM Overview. *Red ones are introduced by TeRM. tLib is TeRM's userspace library. We distinguish tLib on the client and the server by tLib-C and tLib-S respectively.*

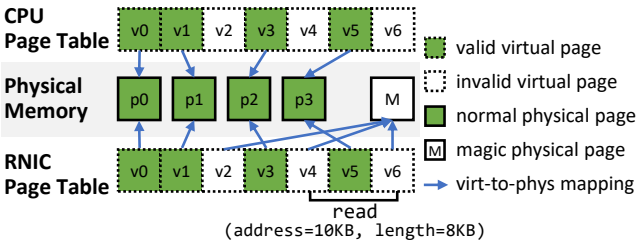

Figure 6: TeRM MR. *We show the RNIC page table of TeRM MR and compare it with the CPU page table. v0 – v6 are virtual pages. p0 – p3 and M are physical pages. The read request starts at the offset of 10KB with a length of 8KB.*

an SSD. TeRM leverages the Linux kernel to do the demand paging between the physical memory and the SSD, and manages the virtual-to-physical mappings in the CPU page table. In this way, the unmodified server-side application can access the virtual memory to maintain in-memory runtime data.

TeRM MR serves remote access from the client-side application. During initialization, the server-side application registers a TeRM MR to expose the virtual memory. tLib-S cooperates with the modified RNIC driver (§5) to manage the RNIC page table of the TeRM MR. Recall *principle #1: onload exception handling from hardware to software*. We orchestrate the RNIC page table and remove RNIC page faults, i.e., the faulting flow (Figure 2) from the TeRM MR.

As illustrated in Figure 6, For all the valid virtual pages (v0, v1, v3, v5), the RNIC page table maps them to normal physical pages (p0 – p3), the same ones that the CPU page table points to. When an RDMA READ accesses valid virtual pages, it retrieves the true data on the correct physical pages. For all the invalid virtual pages (v2, v4, v6), TeRM maps them to one magic physical page (M). TeRM reserves the magic physical page and populates it with a magic pattern. When an RDMA READ accesses invalid virtual pages, the server-side RNIC follows the mapping and retrieves the data on the magic physical page. In this way, the RDMA READ completes normally without triggering the RNIC page fault.

4.1.2 Workflow

Read. A client reads data on the server-side TeRM MR by submitting a read request to tLib-C. The read request describes the addresses of both sides and the length. In Figure 5, tLib-C processes the read request in three steps. ❶ tLib-C generates an RDMA READ according to the user-submitted read request and sends the RDMA READ via the client-side RNIC. ❷ The server-side RNIC returns the data without interacting with the CPU. ❸ tLib-C checks whether the data contains the predefined magic pattern. If no magic pattern is found, all the data are on valid virtual pages. tLib-C has retrieved the valid data and thus completes the read request. In this way, TeRM completes the read request by a one-sided RDMA READ.

If the magic pattern is found, the client determines that it accesses invalid virtual pages and data on these pages are missing. The client fetches the missing data in three steps. ① tLib-C submits an RPC to retrieve the missing data; we call it *RPC READ* hereinafter. ② Receiving an RPC READ, tLib-S reads the missing data to a preallocated and registered bounce buffer. ③ tLib-S returns data to the client. As the bounce buffer has been registered, tLib-S sends the data via RDMA WRITE without triggering RNIC page faults. Afterward, the server notifies the client of the completion. Finally, with all the data fetched, tLib-C completes the read request.

Write. A client writes data to the server-side TeRM MR by submitting a write request to tLib-C. tLib-C processes the write request in three steps like processing a missing read request. ① tLib-C submits an RPC to the server to write the data; we call it *RPC WRITE* hereinafter. ② tLib-S fetches the data from the client to the bounce buffer by RDMA READ. Then it copies the data on the bounce buffer to the virtual memory. ③ tLib-S notifies the client, and tLib-C completes the write request to the application.

4.1.3 Challenges

There are several challenges in the design.

1) As a read request is in byte granularity but the virtual-to-physical mapping of the TeRM MR is in page granularity, precisely identifying invalid virtual pages becomes challenging. We tackle the challenge in a hierarchical manner, from the request level to the page level. We also introduce a set of techniques to reduce network traffic during identification. We detail the design in §4.2.

2) As mentioned in the workflow, TeRM may introduce internal RPCs, i.e., RPC READ and RPC WRITE that access the SSD-extended virtual memory. An intuitive approach is performing `load`/`store`, inducing heavy CPU page faults. Following *principle #2: eliminate CPU page faults from the critical path*, we propose tiering IO. Instead of memory `load`/`store` interfaces, tiering IO resorts to file IO interfaces, i.e., selectively uses buffer IO and direct IO to access the SSD-extended virtual memory. We describe how tiering IO operates at length in §4.3.

3) As TeRM MR and tiering IO eliminate RNIC and CPU page faults from the critical path, it freezes the data placement on the server, unfortunately. A fixed part of the virtual memory is in the physical memory and mapped in the TeRM MR. Without promotion on the critical path by the page faults, a hotspot may be always on the SSD. Facing the challenge, TeRM makes the client and the server collaborate to determine and promote hotspots to physical memory in the background dynamically (§4.4).

4.2 Identifying Invalid Virtual Pages

As a read request is in byte granularity, it leads to two issues during identifying invalid virtual pages, the inter-page issue at the request level and the intra-page issue at the page level. The inter-page issue is that a read request may span multiple virtual pages, some of which are valid but others are invalid. For efficiency, TeRM should identify and fetch only the missing data on invalid virtual pages via RPC. The intra-page issue is that a read request may access only part of a virtual page. TeRM should be able to determine whether a virtual page is valid with any part of the virtual page. Moreover, TeRM should reduce network traffic in identification.

Page division. To tackle the inter-page issue, TeRM adopts page division. It splits the received data at page boundaries into several parts and checks each part separately. Take the read request in Figure 6 as an example. TeRM cuts the data into three parts (on v4 – v6) and checks them one by one.

Byte detection. To tackle the intra-page issue, TeRM adopts byte detection. On the server side, the magic pattern covers all the bytes — not just the beginning or the end — of the magic physical page, e.g., setting every byte to a magic number. On the client side, tLib-C compares the retrieved part of a virtual page with the magic pattern byte by byte. If matched, tLib-C assumes that the part belongs to an invalid virtual page. The read request in Figure 6 accesses the first half of v4 and the last half of v6, which match the magic pattern, so tLib-C determines v4 and v6 are invalid. The data on v5 does not match the magic pattern and thus v5 is valid.

Sparse fetching. After identifying all the invalid virtual pages precisely, tLib-C fetches data sparsely. It submits an RPC READ and tLib-S fetches only the missing data on them. Apart from the addresses of both sides and the access length, the RPC READ also contains a page bitmap to indicate whether each page is valid. With the read request in Figure 6 as an example, the page bitmap is `b'010`, as the first (v4) and the last (v6) virtual pages are invalid. Receiving the RPC READ, tLib-S bypasses all the valid virtual pages. It parses the bitmap to locate and read all the invalid virtual pages.

Combining page division, byte detection, and sparse fetching, TeRM identifies invalid virtual pages of a read request and only fetches the missing data on these pages via RPC. Compared with fetching all the data of a read request, of which only some virtual pages are invalid, TeRM reduces the

amount of data transfer and thus speeds up the miss path.

False positive cases. Identifying invalid virtual pages by the magic pattern may lead to false positive cases. If the server-side application fills a valid virtual page with the magic pattern, the client will determine it as an invalid virtual page falsely. TeRM overcomes the issue with three key insights. First, for random data, the possibility of false positive cases is low. For a single byte (i.e. 8 bits) of random data, the probability is only $1/2^8 = 1/256$. As the data length grows to n bytes, the probability drops exponentially to $1/256^n$, which is negligible. Moreover, TeRM varies the magic pattern dynamically for different processes at different times, to prevent an application from always producing the same data as one specific magic pattern. Finally, even if a false positive case occurs, TeRM handles it as accessing an invalid virtual page and fetches the data again via an RPC READ, without compromising the correctness.

Page bitmap. As tLib-C identifies an invalid virtual page from the magic pattern, RDMA READ for it consumes extra network bandwidth. To reduce the network traffic, we propose a page bitmap to identify an invalid virtual page before RDMA READ. tLib-S maintains a page bitmap for a TeRM MR to indicate whether each virtual page is valid. tLib-C pulls the page bitmap periodically, e.g., per second in our evaluation. For a read request, tLib-C queries the page bitmap first and only sends RDMA READ for valid virtual pages. Afterward, tLib-C submits RPC READ for all invalid virtual pages identified by both the page bitmap and the magic pattern.

Note that the client-side page bitmap may be inaccurate but does not harm read correctness and the overhead is acceptable. If an invalid virtual page is indicated as valid by the bitmap, RDMA READ will return the magic pattern and thus tLib-C can identify it correctly. In contrast, for a valid virtual page indicated as invalid, RPC READ will retrieve the correct data.

One may wonder why we do not use the page bitmap to guide a write request, i.e.,sending RDMA WRITE for a valid virtual page and RPC WRITE for an invalid one. This is because the overhead due to inaccuracy is unacceptable. If an invalid virtual page is indicated as valid, RDMA WRITE on it will trigger an RNIC page fault, which stalls the transmission and consumes no less time than a read-triggered one (hundreds of microseconds as shown in Figure 4).

We discuss the overhead of pulling the page bitmap. With one bit for each 4KB page, the page bitmap size is only 0.003% of the MR. For a 64GB MR, each client pulls 2MB each time, which is negligible against the RNIC bandwidth.

Although TeRM introduces extra network traffic in identifying an invalid virtual page, we argue that the ODP MR also causes additional network traffic due to the RNR NACK. It stalls the QP and wastes more network resources (§3.2).

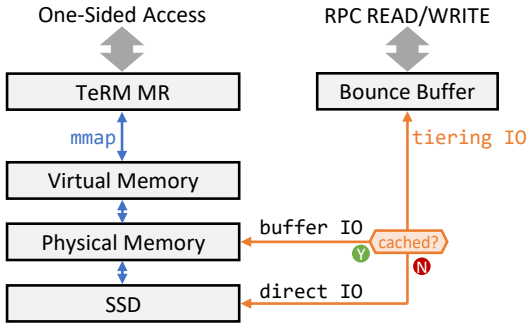

Figure 7: Tiering IO.

4.3 Accessing Data via Tiering IO

As we state in the workflow (§4.1), tLib-S reads and writes the virtual memory to/from the bounce buffer during handling RPC READ/WRITE. Recall that TeRM `mmap`s an SSD to create an area of virtual memory, so that the unmodified server-side application on the CPU can `load`/`store` the SSD-extended virtual memory. Therefore, tLib-S must also access the virtual memory through a kernel-exposed interface, instead of a kernel-unaware interface, e.g., SPDK [7].

Although `memcpy` between the virtual memory and the bounce buffer is an intuitive choice, it triggers heavy CPU page faults (Figure 4) on invalid virtual pages that have been swapped out to the SSD. Following *principle #2: eliminate CPU page faults from the critical path* (§3.2), we propose tiering IO to access the SSD-extended virtual memory, as illustrated in Figure 7. Our key idea is resorting to file IO interfaces instead of memory `load`/`store` interfaces.

Tiering IO orchestrates two interfaces — buffer IO and direct IO — to access different states of virtual pages. Buffer IO invokes `pread`/`pwrite` to access the page cache. Direct IO bypasses the page cache with the `O_DIRECT` flag.

Tiering IO selects the interface to access a virtual page according to its state. 1) If the virtual page resides in the page cache, tiering IO accesses it via buffer IO. The IO can be completed fast by the page cache, without communicating with the SSD. 2) If the virtual page is uncached, accessing the data via buffer IO will incur the page replacement of the page cache. The replacement is time-consuming [5, 8], especially when the page cache is nearly full. Therefore, tiering IO chooses direct IO to bypass the page cache.

We identify three issues to support tiering IO, virtual-to-block mapping, virtual page state, and direct IO granularity. We discuss and tackle them below.

Virtual-to-block mapping. RPC READ/WRITE provides the virtual address to access, we have to convert it to a logical block address (LBA) on SSD for invoking IO. Fortunately, the Linux kernel offers an efficient static virtual-to-block mapping. By `mmap`-ing a given LBA range `[slba, slba + length)` of the SSD, we get a virtual address range `[saddr,`

`saddr + length)`. For a server-side virtual address `addr`, tLib-S calculates its LBA by `addr - saddr + slba`.

Virtual page state. TeRM queries the page cache to learn whether a page is cached. Notably, the page state may be stale by the time invoking the IO call. For example, tiering IO determines that a virtual page is uncached and then accesses it via direct IO, but the page may be cached just before the call begins. We argue that the stale page state does not compromise the correctness, because direct IO read and write flushes and invalidates the page cache respectively, so as to guarantee data consistency [1].

Direct IO granularity. The granularity of RPC READ-/WRITE and direct IO does not match. The former is a byte, while the latter is a block, typically 512B or 4KB. To bridge the granularity gap for RPC READ, we pad offset and length of `pread` to block boundaries. As for RPC WRITE unaligned to a block, we adopt a read-modify-write operation. We use an exclusive lock for each block to control the concurrent read-modify-write operations on the same block.

4.4 Determining and Promoting Hotspots

With the design of TeRM MR and tiering IO, TeRM eliminates RNIC and CPU page faults form the critical path. Although the elimination streamlines the critical path, it freezes the data placement on the server, unfortunately. A fixed part of the virtual memory is in the physical memory and mapped in the TeRM MR. If a hotspot is on the SSD, it will always be accessed by an RPC READ/WRITE with direct IO. Considering the server is unaware of one-sided RDMA accesses from the client, we propose making the client and the server collaborate to determine hotspots and then promote hotspots dynamically, so as to improve the overall performance.

Determining hotspots. TeRM employs client-side tracking and server-side accumulating to count the frequency of read-/write requests and find the hotspots.

TeRM tracks requests at the client, because a hit read request finishes by a one-sided RDMA READ without involving the server-side CPU. tLib-C splits the address space of a TeRM MR at the granularity of a *sample unit* and creates a counter for each unit. When the application submits a read-/write request, tLib-C locates all the requests' spanning sample units and increases their counters. A smaller sample unit results in finer counting, but the TeRM MR is divided into more units and thus the counters occupy a larger memory space. TeRM sets the sample unit to 1MB to achieve the balance between the counting granularity and the counters' memory footprint. With a 32-bit counter for each sample unit, the counters take up only 0.00003% memory space compared to a TeRM MR, which is negligible.

TeRM accumulates counters at the server, given that multiple clients in the cluster may access one TeRM MR. In every *sample period*, tLib-S pulls all the counters from the clients and sums them up. Then it gets a global view of the counters and knows how many times each unit has been accessed in the latest period. A shorter sample period leads to a more timely counting but consumes more network bandwidth during transferring the counters. TeRM sets the sample period to 1 second to balance these two aspects.

The more times that a sample unit is accessed, the hotter TeRM thinks it is. TeRM sorts the units by their counters in descending order and determines the hottest units that can be placed in the physical memory as hotspots.

Promoting hotspots. TeRM promotes hotspots one by one. A unit is skipped if it has been promoted in an earlier period. Otherwise, TeRM invokes the advising flow (Figure 2) to promote the unit. The unit is swapped into the physical memory and mapped in the RNIC page table. Then a later read request on the unit is completed via an RDMA READ and a write one is done via buffer IO write in RPC WRITE.

The advising flow is also time-consuming due to triggering the CPU page fault and updating the RNIC page table. Thus, TeRM does not promote all the hotspots in one promotion. Instead, TeRM promotes as many units as possible within the time of a sample period. Then it begins the next period of determining and promoting hotspots.

The promotion design balances effectiveness and flexibility. If the hotspots remain stable for consecutive periods, TeRM promotes the hottest proportion in the beginning periods and then the less hot ones in the later periods. All the determined hotspots are promoted eventually. However, if the hotspots have changed since the last sample period, promotion for the last period completes fast, and TeRM shifts to promote the latest hotspots immediately. As the promotion is conducted periodically, it occupies little CPU resources.

Consistency discussion. As the promotion and tiering IO in RPC READ/WRITE may access the same page, we discuss the concurrency consistency here. As both trap into the Linux kernel, TeRM reuses the concurrency control in the Linux kernel to guarantee consistency. Note that the promotion always accesses the page cache. If tiering IO performs buffer IO, accesses are routed to the page cache, and hence the concurrency consistency is maintained by the page cache. If tiering IO performs direct IO, read requests do not raise any consistency issue since the SSD always contains the newest version of data. At the beginning and end of write requests in direct IO, the kernel invalidates the page cache so as to prevent old data in the cache and guarantees the consistency.

5 Implementation

We implement TeRM for the Mellanox RNIC. We build the userspace library tLib with about 6,100 lines of C++ code and modify the RNIC driver with about 300 lines of C code.

tLib. It overrides the APIs to manipulate the MR (`ibv_register_mr`) and the RDMA request (`ibv_post_send`, `ibv_poll_cq`). tLib is transparent

to the upper-layer application via `LD_PRELOAD` and has two instances tLib-S and tLib-C.

The server-side application invokes `ibv_register_mr` to register an MR. In the overridden `ibv_register_mr`, tLib-S interacts with the modified RNIC driver to create a TeRM MR and starts an RPC service in the userspace to serve the RPC READ/WRITE from clients. We adopt coroutine in the RPC service and `libaio` to submit direct IO to the SSD asynchronously, so as to enhance the CPU efficiency.

The client-side application calls `ibv_post_send` to submit a read/write request and polls the completion via `ibv_poll_cq`. For a write request, tLib-C converts it to an RPC WRITE in the overridden `ibv_post_send` and submits it to the server. For a read request, tLib-C identifies invalid virtual pages in the overridden `ibv_poll_cq` and submits an RPC READ to the server if necessary.

Considering RDMA requests enjoy low latency, we aim to make tLib execute efficiently. We employ multithreading-friendly and cacheline-aware data structures and mechanisms throughout the implementation, to reduce the extra running overhead introduced by tLib.

RNIC driver. We modify the RNIC driver to support the TeRM MR. We reuse the mechanisms of the ODP MR, including the RNIC page table and the synchronization flows with the CPU page table§2.2. When creating a TeRM MR, the RNIC driver allocates a physical page from the kernel and fills it with the magic pattern. TeRM eliminates the faulting flow as we state in §4 and modifies the invalidation flow in the driver. When the Linux kernel notifies the RNIC driver of invalidating a virtual page, the RNIC driver does not clear the virtual-to-physical mapping as it does for the ODP MR, but instead makes the mapping point to the magic physical page.

6 Evaluation

We evaluate TeRM by microbenchmarks and RDMA-based storage systems to answer the following questions.

- How does TeRM compare with existing approaches? (§6.2)
- How do the design techniques contribute to the end-to-end performance of TeRM? (§6.3)
- How does TeRM perform on dynamic workloads? (§6.4)
- How do workload characteristics affect TeRM? (§6.5)
- How can RDMA-based storage systems benefit from TeRM? (§6.6)

6.1 Experimental Setup

Testbed. We conduct the experiments on a cluster of one server machine and two client machines. The server machine has a 56-core Intel Xeon Gold 6330 CPU, 96GB DRAM, and a 400GB Intel Optane 5800X SSD. The SSD has 1.25/1.16 Mops/s of 4KB random read/write and 4.21/0.69 Mops/s of 512B random read/write. Each client machine has a 36-core Intel Xeon Gold 5220 CPU and 64GB DRAM. We equip the machines with a ConnectX-5 RNIC on each and connect them by a 100Gbps IB RDMA switch.

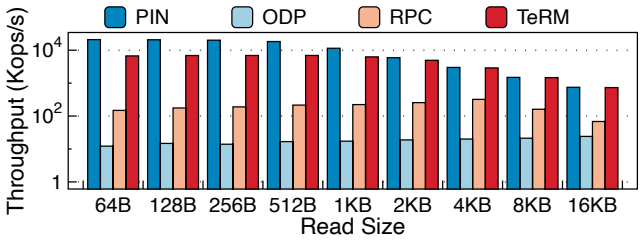

Figure 8: Read Throughput. *The vertical axis is in a logarithmic scale.*

Comparing Targets. We compare TeRM with two approaches ODP and RPC. Moreover, we use PIN to show the ideal upper bound of performance where all data pages are pinned in the physical memory.

- *PIN*. Only in this approach, we do not restrict the available physical memory. The server registers the virtual memory as a pinned MR. The clients read and write the server-side pinned MR by one-sided RDMA READ/WRITE through the original `libibverbs`.
- *ODP*. On the server machine, we register the virtual memory as an ODP MR. The clients also use the original `libibverbs` to submit read/write requests. All the requests are handled by one-sided RDMA READ/WRITE and trigger RNIC page faults on invalid virtual pages.
- *RPC*. All the read/write requests are handled by RPC READ/WRITE. The server-side RPC service accesses the virtual memory via `memcpy` and thus triggers CPU page faults when a virtual page is not mapped.
- *TeRM*. The server registers the virtual memory as a TeRM MR. tLib-C interacts with tLib-S to handle read/write requests submitted by the client, as we describe in the design and implementation.

Workloads. We use a microbenchmark to evaluate the performance. We run it for 60 seconds and report the average throughput. It creates a 64GB virtual memory by `mmap`-ing the SSD on the server machine. We limit the available physical memory to 32GB, 50% size of the virtual memory. The microbenchmark runs 64 client threads, 32 threads on each client machine. Each client thread issues read and write requests to the server, where the accessing positions follow a skewed distribution (Zipfian θ=0.99). For both the RPC approach and TeRM, we create 16 threads for the RPC service on the server machine and bind these threads on eight physical CPU cores. We keep the settings above as the default throughout the experiments unless stated otherwise.

6.2 Overall Performance

In this experiment, we evaluate the read and write performance of access sizes from 64B to 16KB.

	Read 256B		Read 4KB	
	p50 lat.	**p99 lat.**	**p50 lat.**	**p99 lat.**
PIN	3.10	4.21	21.20	24.03
ODP	3.03	29,103.92	4.60	45,781.38
RPC	38.55	109.79	26.01	93.62
TeRM	3.50	30.07	16.51	52.02

Table 1: Read Latency (μs)

6.2.1 Read

We report read throughput in Figure 8 and latency in Table 1. We analyze the performance of TeRM against ODP, RPC, and PIN respectively in the following.

TeRM vs. ODP. The throughput of TeRM and ODP achieves 6.93Mops/s and 24.09Kops/s respectively. TeRM outperforms ODP by 30.46× – 549.63×. ODP has the lowest p50 latency of all four approaches. This is because most read requests are hit in the physical memory. The RNIC on the ODP approach is the least utilized in transferring data and thus shows the lowest latency to finish a hit RDMA READ. However, the long p99 latency demonstrates that ODP suffers from heavy RNIC and CPU page faults on the miss path. TeRM reduces the p99 latency by up to 967.74×, thanks to the much more efficient miss path.

Notably, PART [32], a hardware solution like the ODP MR, reports a 31μs latency of a faulting RDMA request, which is lower than ODP in our experiment. The latency difference mainly arises from the fact that PART evaluates a minor page fault while we evaluate a major one. A major page fault has to load data from the SSD. Nevertheless, TeRM incurs a lower average latency of 26.61μs for a missed read request. This is because TeRM leveragers tiering IO to build an efficient miss path that bypasses the CPU page fault.

TeRM vs. RPC. The RPC approach reaches a throughput of 320.75Kops/s. It does not provide the maximum throughput, because the RPC approach triggers CPU page faults when pages are not mapped. The CPU page fault performs poorly and does not scale well with more server threads, which is also reported by previous studies [11, 21, 28–30]. The RPC approach performs better than ODP because clients use two-sided RDMA primitives to submit read requests. In this way, the RPC approach avoids RNIC page faults. TeRM surpasses the throughput of the RPC approach by 9.05× – 45.19×. It decreases the p50 and p99 latency by 11.02× and 1.57× respectively. TeRM has significant improvement because it utilizes the server CPU more efficiently in two aspects. First, TeRM tries one-sided RDMA READ and only handles the missed read requests via the RPC service on the server CPU, but the RPC approach involves the server CPU in processing all the read requests. Moreover, TeRM proposes tiering IO to avoid CPU page faults, while the RPC approach may still trigger CPU page faults during memcpy.

TeRM vs. PIN. When the read size is less than 1KB, the throughput of TeRM is stable around 6.86Mops/s, achieving up to 37.79% of the PIN throughput. With 256B as an example, the hit ratio of read requests is 69.44%, and the latency is as low as PIN. The slowdown of TeRM mainly arises from the missed read requests, each of which costs about 19.26μs. In this scenario, the RPC service becomes the bottleneck of the overall performance.

For large read requests above 1KB, TeRM greatly narrows the gap with PIN. It achieves 54.55% throughput of PIN at 1KB and the ratio goes up to 96.71% at 4KB. The narrowing gap results from the shrinking latency difference between hit (16.51μs) and missed (26.61μs) read requests. In this case, TeRM saturates the RNIC bandwidth.

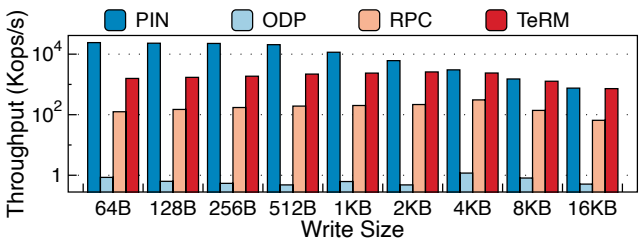

Figure 9: Write Throughput.*The vertical axis is in a logarithmic scale.*

	Write 256B		Write 4KB	
	p50 lat.	**p99 lat.**	**p50 lat.**	**p99 lat.**
PIN	2.91	3.89	21.29	23.40
ODP	56,158.64	71,006.89	19,884.87	41,636.82
RPC	43.89	131.52	317.12	1,772.35
TeRM	15.22	117.27	21.18	59.73

Table 2: Write Latency (μs)

6.2.2 Write

We show write throughput in Figure 9 and latency in Table 2. We compare TeRM with ODP, RPC, and PIN.

TeRM vs. ODP. TeRM and ODP have throughput up to 2.58Mops/s and 1.18Kops/s respectively. We output the throughput second by second and find that ODP is unstable and jitters sharply. It reaches a peak throughput of 4.28Kops/s at some time but may also stall for more than a second. Nevertheless, TeRM surpasses ODP's peak throughput by up to 1,195.81×. ODP performs worse on write than read, because write incurs higher swapping overhead. To swap out a read-only page, the OS kernel just drops it from the physical memory because it is clean. But to swap out a written page, the OS kernel has to write the dirty page back to the SSD.

TeRM vs. RPC. The RPC approach reaches a throughput of 308.34Kops/s. Even though TeRM also handles write requests via RPC WRITE, it outperforms the RPC approach by up to 12.60×. TeRM reduces the p50 and p99 latency by 14.97× and 29.67×. The results demonstrate the effectiveness of tiering IO and promoting hotspots compared to `memcpy`.

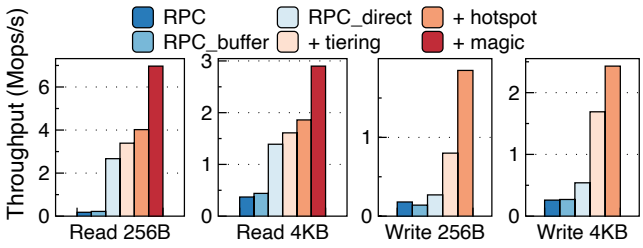

Figure 10: Contribution of Each Technique.

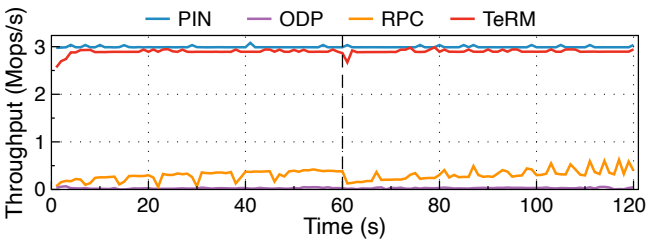

Figure 11: Performance of Dynamic Workloads. *We change the hotspots at the 60th second.*

TeRM vs. PIN. When the write size is smaller than 512B, the throughput is around 1.71Mops/s. Taking 256B as an example, 72.53% of write requests hit the page cache and finished by buffer IO. The rest of write requests are completed by the read-modify-write operation as we describe in §4.3, which limits the overall performance. When the size grows to 512B, the write throughput climbs to 2.19Mops/s. This is because tiering IO writes uncached data simply by a direct IO write rather than the time-consuming read-modify-write. The SSD can provide 694Kops/s of 512B write operations and becomes the bottleneck. As the write size goes up, the SSD's throughput increases and so does the write throughput of TeRM. TeRM reaches the throughput of 2.38Mops/s at 4KB, 78.69% of the PIN throughput, where the SSD exhibits 1.16Mops/s of 4KB write. When the write size is even larger, TeRM further reduces its gap with the PIN approach. TeRM achieves 727.28Kops/s throughput at 16KB, 96.32% of the PIN approach. In this scenario, the RNIC bandwidth dominates the overall performance.

6.3 Contribution of Each Technique

In this experiment, we analyze how each technique contributes to TeRM, as reported in Figure 10. We choose 256B and 4KB as representatives of small and large read/write sizes. We use three baselines, RPC, *RPC_buffer*, and *RPC_direct*. RPC_buffer and RPC_direct are the same as the RPC approach, except that we replace the server-side `memcpy` with buffer IO and direct IO respectively. We introduce these two baselines to study the advantage of tiering IO compared with existing IO interfaces. Then we gradually enable tiering IO (§4.3), promoting hotspots (§4.4), and TeRM MR (§4.1&§4.2) atop RPC_direct. Three techniques are annotated as *+tiering*, *+hotspot*, and *+magic* in Figure 10. We test Read 4KB using eight RPC threads, which are enough for TeRM in this case; we analyze server-side CPU usage in more detail later in §6.5.4. We apply the TeRM MR last because it can function better when hotspots are promoted. Notably, the TeRM MR does not apply to write requests.

The experimental results demonstrate that all techniques contribute to the performance improvement of TeRM.

Baselines. We first compare three baselines. RPC_buffer performs as poorly as RPC. It avoids CPU page faults but cannot eschew the heavy page replacement [5, 8]. RPC_direct surpasses RPC_buffer by 1.90× – 12.05×. It bypasses the page cache but loses the opportunity to access cached data fast.

+tiering. Tiering IO improves the performance by 1.16× – 3.10× over RPC_direct. It accesses data via the page cache whenever possible. When the data is on the SSD, tiering IO accesses it directly through the device. In this way, tiering IO manages to exploit the high-performance physical memory and avoid the heavy page cache maintenance simultaneously.

+hotspot. Determining and promoting hotspots further increases the throughput by 1.16× – 2.31×. With the hotspots promoted in the physical memory, tiering IO completes more hot data requests from the page cache.

+magic. TeRM MR raises the throughput by 1.56× – 1.73×. The hit read requests are handled through one RDMA READ operation without bothering the server-side CPU. As the RPC service only handles miss read requests instead of all read requests, TeRM utilizes both the RNIC and the server-side CPU more efficiently.

More specifically, the page bitmap also plays a remarkable role in performance improvement. The hit ratio of read 4KB requests is about 73%. Without the page bitmap, the remaining 27% read requests are transferred twice, the first time via RDMA READ and the second time via RPC READ. The end-to-end throughput is 2.37Mops/s, only 78.97% of the PIN approach. With the page bitmap, less than 0.1% read requests are transferred twice. The throughput is 2.90Mops/s, achieving 96.71% of the PIN approach.

6.4 Dynamic Workloads

We evaluate how TeRM reacts to dynamic hotspots and plot the results in Figure 11. We run the benchmark for 120 seconds and change the hotspots at the 60th second. We have two observations from the results. 1) TeRM performs more stably than ODP and RPC. TeRM is stable at 2.89Mops/s and drops by only 6.82% after switching hotspots. The throughput of ODP and RPC jitters sharply and drops by 1.77× and 3.03× after the switching. 2) TeRM determines and promotes hotspots effectively and efficiently. The throughput of TeRM returns to the peak quickly in one second, while it takes ODP

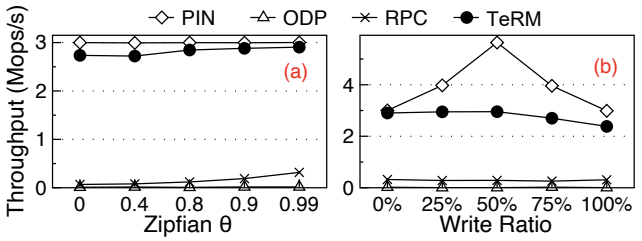

Figure 12: Performance with Varying (a) Skewness and (b) Write Ratios.

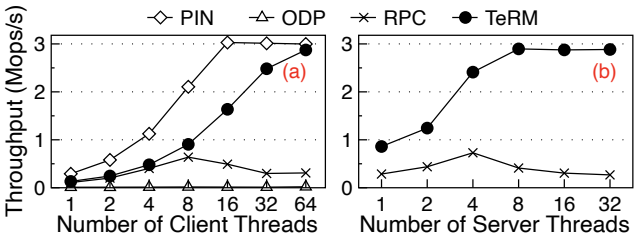

Figure 13: Performance with Different Numbers of (a) Client Threads and (b) Server Threads.

and RPC six seconds.

6.5 Sensitivity Analysis

We evaluate how the characteristics of workloads impact the performance of TeRM. We show the read performance of 4KB in these experiments by default, unless otherwise stated.

6.5.1 Skewness

Figure 12(a) plots the performance of approaches with varying skewness. For the uniform distribution ($\theta = 0$), TeRM exhibits more significant improvement against existing approaches, compared with the skewed distribution. It outperforms ODP and RPC by 265.24× and 40.40× respectively, achieving 91.22% of PIN. The hotspots are more concentrated when θ increases. The PIN approach is stable with varying skewness. ODP, RPC, and TeRM show higher throughput as the skewness grows, because more requests are within the hotspots that reside in the physical memory.

6.5.2 Write Ratio

Figure 12(b) depicts the throughput with five different write ratios, 0% (read-only), 25% (read-most), 50% (read-write), 75% (write-most), and 100% (write-only). The PIN approach shows improvement on read-write-mixed requests because it exploits the full-duplex performance of the RDMA network. As for TeRM, mixing read and write slows down the direct IO performance and restricts the overall throughput. ODP and RPC do not perform better for read-write-mixed requests compared with the read-only or write-only scenario.

6.5.3 Client Threads

Figure 13(a) reports the throughput with different numbers of client threads. As the number of client threads goes up, the throughput of TeRM grows linearly and reaches 2.48Mops/s at 32 threads. The throughput of PIN also increases linearly and hits the peak at 16 threads. ODP and RPC scale poorly with the increasing number of client threads.

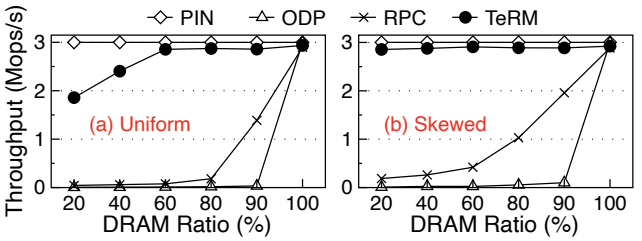

Figure 14: Performance with Varying DRAM Ratios of (a) Uniform and (b) Skewed Workloads.

6.5.4 Server Threads

Figure 13(b) shows the throughput with different numbers of server threads for the RPC service. TeRM scales with the increasing number of server threads and reaches the peak at eight threads. The RPC approach does not scale well because the CPU page fault scales poorly [30]. TeRM outperforms the RPC approach by 2.84×– 10.76×. Even with one server thread, TeRM exceeds the peak throughput of the RPC approach. The results demonstrate the CPU efficiency of TeRM.

6.5.5 DRAM Ratio

As shown in Figure 14, we evaluate the performance with different sizes of DRAM, i.e., available physical memory. TeRM performs well with varying DRAM sizes. Even with only 20% DRAM, TeRM provides 95.10% and 61.93% throughput of the PIN approach on skewed and uniform workloads. It outperforms ODP and RPC by up to 388.29× and 41.78×. The enhancement is higher compared with the default 50% DRAM setting in our experiments. This demonstrates that TeRM still acts efficiently under a low DRAM ratio. All approaches have higher throughput with more DRAM. With the 90% DRAM ratio, the RPC approach increases to 1.95Mops/s.

It is worth noting the performance with a 100% DRAM ratio, where all data fits in the physical memory. This scenario shows the extra overhead of each approach. Compared with PIN, TeRM introduces 2.63% overhead, which is negligible. ODP and RPC also exhibit performance close to PIN.

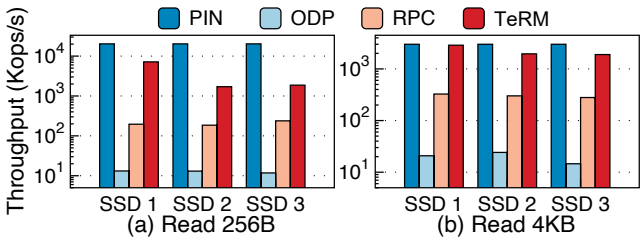

Figure 15: Performance with Different SSDs of (a) Read 256B and (b) Read 4KB. *The vertical axis is in a logarithmic scale. SSD 1: Intel Optane P5800X. SSD 2: Intel Optane P4800X. SSD 3: Samsung PM9A3. More details about SSDs are listed in Table 3.*

ID	Product Model	Read 512B	Read 4KB
SSD 1	Intel Optane P5800X [4]	4,216	1,255
SSD 2	Intel Optane P4800X [3]	586	586
SSD 3	Samsung PM9A3 [6]	600	619

Table 3: Throughput of Different SSDs (Kops/s). *We test their random throughput on a 64GB area using `fio` with 16 threads and libaio (queue depth = 4). SSD 1 & 2 use Intel Optane memory as the storage media. SSD 3 uses NAND flash as the storage media.*

6.5.6 SSD

We evaluate how different SSDs impact the performance and plot the results in Figure 15. The details of the SSDs are listed in Table 3. TeRM running on SSD 2 and SSD 3 achieves close throughput at about 1.95Mops/s. It outperforms ODP and RPC by up to 158.83× and 9.22×. SSD 2 and SSD 3 have similar IO throughput and limit the overall performance. The experimental results show that TeRM acts effectively on different SSDs with different types of storage media.

6.6 RDMA-based Storage Systems

We evaluate how existing RDMA-based storage systems can benefit from TeRM. We choose an RDMA-based file system, Octopus [25], and an RDMA-based key-value system, XStore [37]. We keep the programs unmodified, except `mmap`-ing the SSD to get a large area of virtual memory and registering it as a pinned, ODP, or TeRM MR in their initialization stage.

6.6.1 Octopus: A File System

Octopus is an RDMA-based file system. The server initializes a large area of virtual memory to store metadata and data, and exposes it via an MR. Meanwhile, it runs an RPC service for processing metadata. During accessing a file, the client first communicates with the server via the metadata RPC, to retrieve metadata of a file, e.g., data addresses. Then it reads or writes the server-exposed MR to access file data.

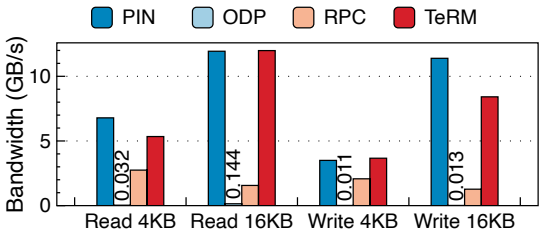

Figure 16: Octopus Performance.

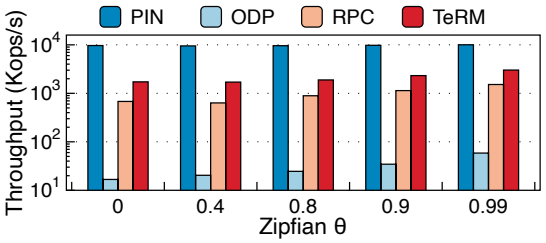

Figure 17: XStore Performance. *The vertical axis is in a logarithmic scale. We use the YCSB-C workload with 8B keys and 128B values.*

On the server machine, we boot 16 threads for metadata service on 16 cores. We run 32 client processes. Each of them reads/writes 4KB/16KB on a 1GB file, where the access positions follow a skewed distribution (Zipfian $\theta = 0.99$). The metadata and data occupy about 35GB of virtual memory on the server; we limit the available physical memory to 18GB.

Figure 16 reports the results. TeRM achieves 82.99–642.23× ODP and 1.77–7.68× RPC. It performs almost the same as PIN on Read 16KB and Write 4KB. Accessing 4KB is slower than 16KB because the client fetches metadata before transferring data. In this scenario, the metadata service bottlenecks the throughput.

6.6.2 XStore: A Key-Value System

XStore is an RDMA-based key-value system. The server maintains a B+ tree and trains a learned index on the virtual memory. It exposes the virtual memory via an MR. The client leverages the learned index to predict the value's address and reads the server-side MR to `get` it. XStore handles `put` operations via RPC. The server runs an RPC service to process `put` requests from the client.

In our experiment, the server initializes a B+ tree containing 8B keys and 128B values. XStore occupies 32GB of virtual memory on the server and we limit the available physical memory to 16GB. Since `put` operations are based on XStore's own RPC, we evaluate how TeRM benefits the `get` performance. We use a YCSB-C workload and vary the skewness of the keys' distribution.

Figure 17 shows the experimental results. TeRM outperforms the ODP and RPC approach by up to 102.97× and 2.69× respectively. As the skewness increases, `get` through-

put increases because hotspots are more concentrated in the physical memory. TeRM achieves 30.07% throughput of the PIN approach at Zipfian $\theta = 0.99$.

The experiments of Octopus and XStore show that RDMA-based storage systems can gain significant performance enhancement from TeRM compared to the ODP and RPC approaches. TeRM saves physical memory and achieves comparable performance against the PIN approach.

7 Related Work

Extending local memory. With the advent of high-performance SSD and network, a host of works focus on extending local memory with the SSD or remote memory in recent years. They extend local memory from different levels, the application programming level [33,35], the virtual address level [11,18,21,28–30,42], and the hardware level [9,13,31]. Then the application process can run on a memory space larger than the physical memory and swap memory pages to the SSD or remote memory.

TeRM differs from these works in target problems and applications. These works focus on extending the private virtual memory of a CPU process and optimizing CPU page faults. Local memory is not exposed and only accessible by the process. Therefore, they target applications like in-memory graph processing systems (e.g., PowerGraph [17]) and big data systems (e.g., Spark [43]). TeRM extends the RDMA-attached memory exposed by the RNIC and tackles RNIC page faults. The memory is shared in the cluster and can be *concurrently* accessed by the server (via CPU) and multiple clients (via the RNIC). TeRM mainly aims at RDMA-based storage systems, e.g., Octopus [25] and XStore [37] in our evaluation.

ODP MR and RNIC page fault. Lesokhin et al. introduce ODP MR and page fault support for the RNIC [22], so that initializing an MR need not pin pages in physical memory. PART [32] also builds a mechanism to handle RNIC page faults on a prototype hardware platform. These works handle the exception in the hardware and thus are restricted by the limited hardware resources. TeRM proposes onloading exception handling from hardware to software.

Onloading from RNIC. Researchers from system and network communities also propose onloading functionalities from the RNIC to the CPU. For example, FaSST [20] and eRPC [19] reimplement reliability on the CPU, to address the scalability issues of the RC connection. Flor [24] onloads flow control from the RNIC to the CPU to support heterogeneous RNIC deployment. In contrast, TeRM targets page fault.

8 Conclusion

We present TeRM in this paper, an efficient approach to extending RDMA-attached memory with SSD. It onloads exception handling (i.e., RNIC page fault) from hardware to software. The experimental results on the microbenchmark and unmodified RDMA-based storage systems demonstrate the effectiveness of TeRM.

Acknowledgements

We sincerely thank our shepherd Joo-young Hwang for helping us improve the paper. We also thank the anonymous reviewers for their feedback. This work is supported by the National Key R&D Program of China (Grant No. 2021YFB0300500), the National Natural Science Foundation of China (Grant No. U22B2023 & 61832011), and the China Postdoctoral Science Foundation (Grant No. 2022M721828).

References

[1] Direct IO and page cache. https://lists.kernelnewbies.org/pipermail/kernelnewbies/2013-July/008660.html.

[2] Heterogeneous Memory Management (HMM) - Linux Kernel Documentation. https://www.kernel.org/doc/html/v5.19/vm/hmm.html.

[3] Intel Optane SSD P4800X. https://www.intel.com/content/www/us/en/products/sku/97161/intel-optane-ssd-dc-p4800x-series-375gb-2-5in-pcie-x4-3d-xpoint/specifications.html.

[4] Intel Optane SSD P5800X. https://www.intel.com/content/www/us/en/products/sku/201861/intel-optane-ssd-dc-p5800x-series-400gb-2-5in-pcie-x4-3d-xpoint/specifications.html.

[5] The page cache and page writeback. http://books.gigatux.nl/mirror/kerneldevelopment/0672327201/ch15.html.

[6] Samsung PM9A3 SSD. https://semiconductor.samsung.com/ssd/datacenter-ssd/pm9a3/.

[7] Storage Performance Development Kit. https://spdk.io/.

[8] A parallel page cache: IOPS and caching for multicore systems. In *4th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 12)*, Boston, MA, June 2012. USENIX Association.

[9] Ahmed Abulila, Vikram Sharma Mailthody, Zaid Qureshi, Jian Huang, Nam Sung Kim, Jinjun Xiong, and Wen-mei Hwu. Flatflash: Exploiting the byte-accessibility of ssds within a unified memory-storage hierarchy. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 971–985, 2019.

[10] Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra J. Marathe, Athanasios Xygkis, and Igor Zablotchi. Microsecond consensus for microsecond applications. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 599–616. USENIX Association, November 2020.

[11] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Can far memory improve job throughput? In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[12] Thomas E. Anderson, Marco Canini, Jongyul Kim, Dejan Kostić, Youngjin Kwon, Simon Peter, Waleed Reda, Henry N. Schuh, and Emmett Witchel. Assise: Performance and availability via client-local NVM in a distributed file system. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1011–1027. USENIX Association, November 2020.

[13] Duck-Ho Bae, Insoon Jo, Youra Adel Choi, Joo-Young Hwang, Sangyeun Cho, Dong-Gi Lee, and Jaeheon Jeong. 2B-SSD: the case for dual, byte-and block-addressable solid-state drives. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 425–438. IEEE, 2018.

[14] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. FaRM: Fast Remote Memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, page 401–414, USA, 2014. USENIX Association.

[15] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: Distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 54–70, New York, NY, USA, 2015. Association for Computing Machinery.

[16] Takuya Fukuoka, Shigeyuki Sato, and Kenjiro Taura. Pitfalls of infiniband with on-demand paging. In *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 265–275, 2021.

[17] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed Graph-Parallel computation on natural graphs. In *10th USENIX symposium on operating systems design and implementation (OSDI 12)*, pages 17–30, 2012.

[18] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G Shin. Efficient memory disaggregation with infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 649–667, 2017.

[19] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter RPCs can be general and fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 1–16, Boston, MA, February 2019. USENIX Association.

[20] Anuj Kalia, Michael Kaminsky, and David G. Andersen. FaSST: Fast, scalable and simple distributed transactions with Two-Sided (RDMA) datagram RPCs. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 185–201, Savannah, GA, November 2016. USENIX Association.

[21] Gyusun Lee, Wenjing Jin, Wonsuk Song, Jeonghun Gong, Jonghyun Bae, Tae Jun Ham, Jae W. Lee, and Jinkyu Jeong. A case for hardware-based demand paging. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 1103–1116, 2020.

[22] Ilya Lesokhin, Haggai Eran, Shachar Raindel, Guy Shapiro, Sagi Grimberg, Liran Liss, Muli Ben-Yehuda, Nadav Amit, and Dan Tsafrir. Page fault support for network controllers. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, page 449–466, New York, NY, USA, 2017. Association for Computing Machinery.

[23] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. Pond: Cxl-based memory pooling systems for cloud platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, pages 574–587, New York, NY, USA, 2023. Association for Computing Machinery.

[24] Qiang Li, Yixiao Gao, Xiaoliang Wang, Haonan Qiu, Yanfang Le, Derui Liu, Qiao Xiang, Fei Feng, Peng Zhang, Bo Li, Jianbo Dong, Lingbo Tang, Hongqiang Harry Liu, Shaozong Liu, Weijie Li, Rui Miao, Yaohui Wu, Zhiwu Wu, Chao Han, Lei Yan, Zheng Cao, Zhongjie Wu, Chen Tian, Guihai Chen, Dennis Cai, Jinbo Wu, Jiaji Zhu, Jiesheng Wu, and Jiwu Shu. Flor: An open high performance RDMA framework over heterogeneous RNICs. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI*

23), pages 931–948, Boston, MA, July 2023. USENIX Association.

[25] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. Octopus: an RDMA-enabled distributed persistent memory file system. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 773–785, Santa Clara, CA, July 2017. USENIX Association.

[26] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using One-Sided RDMA reads to build a fast, CPU-Efficient Key-Value store. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 103–114, San Jose, CA, June 2013. USENIX Association.

[27] Christopher Mitchell, Kate Montgomery, Lamont Nelson, Siddhartha Sen, and Jinyang Li. Balancing CPU and network in the cell distributed B-Tree store. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 451–464, Denver, CO, June 2016. USENIX Association.

[28] Anastasios Papagiannis, Manolis Marazakis, and Angelos Bilas. Memory-mapped I/O on steroids. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 277–293, 2021.

[29] Anastasios Papagiannis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. An efficient memory-mapped key-value store for flash storage. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 490–502, 2018.

[30] Anastasios Papagiannis, Giorgos Xanthakis, Giorgos Saloustros, Manolis Marazakis, and Angelos Bilas. Optimizing memory-mapped I/O for fast storage devices. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*, pages 813–827, 2020.

[31] Christian Pinto, Dimitris Syrivelis, Michele Gazzetti, Panos Koutsovasilis, Andrea Reale, Kostas Katrinis, and H Peter Hofstee. Thymesisflow: A software-defined, hw/sw co-designed interconnect stack for rack-scale memory disaggregation. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 868–880. IEEE, 2020.

[32] Antonis Psistakis, Nikos Chrysos, Fabien Chaix, Marios Asiminakis, Michalis Giannioudis, Pantelis Xirouchakis, Vassilis Papaefstathiou, and Manolis Katevenis. Part: Pinning avoidance in rdma technologies. In *2020 14th IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*, pages 1–8, 2020.

[33] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K Aguilera, and Adam Belay. AIFM:High-Performance,Application-Integrated far memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 315–332, 2020.

[34] Alex Shamis, Matthew Renzelmann, Stanko Novakovic, Georgios Chatzopoulos, Aleksandar Dragojević, Dushyanth Narayanan, and Miguel Castro. Fast general distributed transactions with opacity. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, pages 433–448, New York, NY, USA, 2019. Association for Computing Machinery.

[35] Suhas Jayaram Subramanya, Harsha Vardhan Simhadri, Srajan Garg, Anil Kag, and Venkatesh Balasubramanian. BLAS-on-flash: An efficient alternative for large scale ML training and inference? In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 469–484, 2019.

[36] Konstantin Taranov, Salvatore Di Girolamo, and Torsten Hoefler. CoRM: Compactable Remote Memory over RDMA. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 1811–1824, New York, NY, USA, 2021. Association for Computing Machinery.

[37] Xingda Wei, Rong Chen, and Haibo Chen. Fast RDMA-based ordered Key-Value store using remote learned cache. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 117–135. USENIX Association, November 2020.

[38] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. Deconstructing RDMA-enabled distributed transactions: Hybrid is better! In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 233–251, Carlsbad, CA, October 2018. USENIX Association.

[39] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, and Dimitrios Skarlatos. Tmo: Transparent memory offloading in datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, pages 609–621, New York, NY, USA, 2022. Association for Computing Machinery.

[40] Bin Yan, Youyou Lu, Qing Wang, Minhui Xie, and Jiwu Shu. Patronus: High-Performance and Protective Remote Memory. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 315–330, Santa Clara, CA, February 2023. USENIX Association.

[41] Jian Yang, Joseph Izraelevitz, and Steven Swanson. Orion: A distributed file system for Non-Volatile main

memory and RDMA-Capable networks. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 221–234, Boston, MA, February 2019. USENIX Association.

[42] Wonsup Yoon, Jisu Ok, Jinyoung Oh, Sue Moon, and Youngjin Kwon. Dilos: Do not trade compatibility for performance in memory disaggregation. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, page 266–282, New York, NY, USA, 2023. Association for Computing Machinery.

[43] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 10)*, 2010.

A Artifact Appendix

Abstract

The artifact provides implementation source code and evaluation scripts of TeRM. It overloads the APIs of `libibverbs` and can be integrated with an existing RDMA application transparently by `LD_PRELOAD`.

Scope

The artifact helps understanding our design and implementation details better, including those that we do not mention in the paper due to space limit. It allows to reproduce the experimental results in the paper. It also provides examples for developers to integrate TeRM with their RDMA applications.

Contents

The implementation source code in the artifact contains two parts, the userspace shared library `libterm` (tLib) and the modified RNIC driver. Moreover, the artifact provides evaluation scripts and the third-party applications, including XStore and Octopus.

Hosting

The artifact is available at https://github.com/thustorage/TeRM. The `main` branch has the latest contents.