BlueTAS - Offloading TAS on the BlueField SmartNIC

CS 395T Datacenters - Project Report

Ishank Arora **Sai Surya Duvvuri** **Saumya Prakash** **Divyanshu Saxena** **William Zhang**

The University of Texas at Austin

Abstract

An increase in the capabilities of SmartNICs have lead to new opportunities in offloading networking related computations from CPUs. This frees up CPUs to serve more application related computations, a desirable quality in datacenters. An increasing array of jobs are being offloaded to the SmartNICs in order to ensure CPUs have more time to focus on application workloads instead of networking and security. TAS, a TCP acceleration service for RPCs is one such architecture that could benefit from offloading its computations to the SmartNICs. In this paper, we explore this concept and implement a working prototype of TAS, a TCP Acceleration Service, on a Mellanox BlueField SmartNIC.

1 Introduction

For datacenter stacks, TCP packet processing efficiency is extremely crucial - especially given the fact that network speeds are rising at a much faster rate than the CPU speeds (which are practically stagnant). Previous works in this direction have tried to deal with this discrepancy by either kernel-bypass mechanisms [3, 6], or by separating the TCP stack into optimized fast path for packet delivery and a slow path for uncommon case operations [4].

At the same time, recent works aiming to accelerate services that are close to the networking stack. These services include load balancing [1], network functions [7] and other distributed applications in general [5]. In this work, we explore the design space of offloading TAS, a TCP acceleration service on a SmartNIC.

1.1 Background

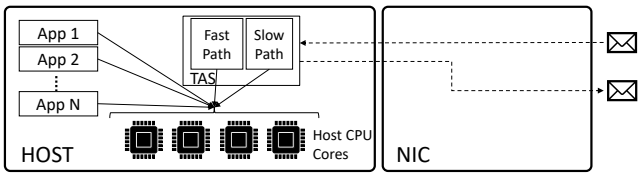
TAS:

TAS is a service that for TCP acceleration to optimize for remote procedure calls (RPCs) by bypassing the kernel [4]. It runs as a user level service which enables efficiency and easy development. There are two main components to TAS: the fast path which takes care of common-case packet processing for RPCs, and the slow path which deals with handling the set up and teardown of connections, congestion control, and timeouts. These components are run as separate threads in user-space.

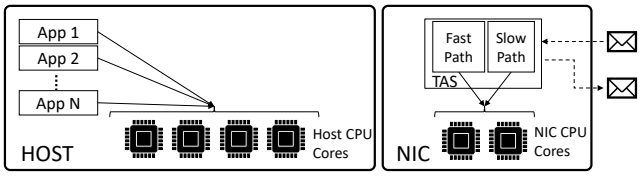
The goal of this separation is to dedicate entire CPUs to TCP stack processing. TAS achieves high efficiency by running dedicated fast path on dedicated CPU cores. This requires large amounts of CPU processing, and we want these CPUs to be available for running core application logic rather than processing network packets. The way that TAS interacts with applications is from both the fast path and the slow path using shared memory. The fast path communicates with the application for data packet payloads, and the slow path communicates for connection set up or tear down. Communication between the slow path and the fast path consists of handling congestion statistics, retransmissions, and control packets.

SmartNICs:

SmartNICs are an emerging type of programmable NICs (Network Interface Controllers) that comes with additional computing resources - especially multicore-SOC. These have recently emerged to close the gap between precious CPU computing power and the increased demand on network bandwidth. There's a lot of research that goes into using these additional computing resources to offload some of the CPU load



Applications and TAS on the Host



Applications on Host and TAS on the SmartNIC

Figure 1: Motivation for offloading TAS to SmartNIC

regarding networking and security such as encryption/decryption, firewall, TCP/IP and HTTP processing.

An advantage of offloading processing to SmartNICs is that since the SmartNICs are designed with simple architectures, they are much more cost-effective- allowing the datacenter to expand its capacity at a low cost [5]. CPU offload is one of the best potentials uses of a SmartNIC.

For this project, we were provided access to a Mellanox BlueField SmartNIC. It is an ARM-based machine and is capable of running a full Linux installation.

1.2 Motivation

With the CPU speeds lagging behind network speeds in datacenters, running any network stack on host CPUs will affect the number of network heavy application instances that host can handle. TAS addresses this problem by introducing fast path for common case RPC calls in datacenters. But, TAS still uses host CPU cores to process packets. Essentially implying that a subset of the host cores are dedicated for running the TAS fast path. Offloading TAS computations to SmartNIC cores as shown in Figure 1 will open up room for more processing for the applications in the host.

2 Design

It follows that from our desire to save CPU resources, we offload the slow and fast paths to the SmartNIC. We first list down the major challenges in enabling such an offload:

#1: Clear separation of roles of the host and the SmartNIC: Driven by the goal to offload the TAS stack on the SmartNIC, a major challenge was to decide what fraction of the TAS stack should be run on the host and what parts should be run on the SmartNIC. Crucially, understanding the resource requirements of the TAS fast path, slow path and the application interface; was important to come up with an efficient role division.

#2: Communication mechanism between the host and the SmartNIC: The foremost challenge in offloading TAS onto the SmartNIC is that the fast path and slow path interact with the application using shared memory queues. Offloading TAS to the SmartNIC, implied that one or more of these components shall not be able to use shared memory on the host. Hence, a **low-overhead, efficient** communication mechanism was needed for the host-NIC communication. Crucially, it was important to keep the host CPU cores free of processing as much as possible.

From the design of TAS we know that the slow path runs as a separate thread within the TAS process. And the fast path is run on multiple dedicated cores. So first and foremost, it is essential that we offload the fast path. By doing so, we ensure that the relevant CPU resources free up. And so, we begin our design journey by considering what it would mean to offload the fast path.

The original TAS implementation relies on the functionality of a shared memory hugepage region. By offloading the fast path alone, we offload the most CPU intensive task. In doing so, we break the shared memory infrastructure that TAS is built on. To solve this, we decide to take advantage of the RDMA functionality available between the SmartNIC and the host for communication. Of course, this comes with a trade-off. By taking a hit on the time it takes for communicating between the fast path and TAS, we reclaim multiple cores on the host, whose job was to only serve the fast path.

The fast path and slow path, together, perform the task of a typical TCP implementation. The fast path takes care of common cases, and the slow path takes care of uncommon cases (for example, congestion control, retransmissions). Congestion control is central in TCP's functionality, and so we decided having

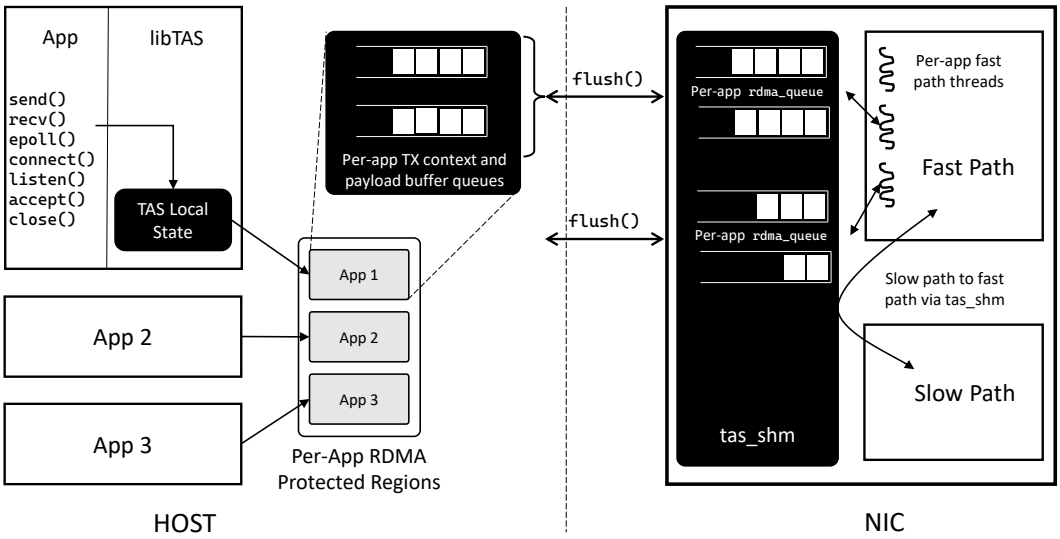


Figure 2: System Design

the slow path and fast path separated on two machines would not be conducive to the speed and efficiency of TAS. Hence, in our final design, we ensure that the slow and fast path are housed on the SmartNIC. This allows us to maintain the shared memory infrastructure between the slow path and the fast path, and thus retain the advantages of the shared memory access between the slow path and fast path communication and gain the advantages of offloading fast path (and slow path too) from the cores of the host to the cores of the SmartNIC.

Figure 2 summarises how we implement our proposed changes. On the application side, we modify LibTAS to ensure that whenever there is communication between the application and the slow path or between the application and the fast path, LibTAS uses RDMA to communicate. It is important to note that we do not directly communicate to the slow path or the fast path. Instead, RDMA calls between the application and the slow and fast paths takes place between the memory regions of the application and the shared memory allocated on the NIC. So, for every application, we assign it a protected local memory region through which it can receive the data in the queues it has between itself and the slow and fast paths. This protected region is required for RDMA [2]. Similarly, it is also required on the other end of the connection, i.e. on the NIC, and that's why the shared memory region on the SmartNICis also protected. Marking a region for protection is an expensive process. So we decide to do protect memory per application at initialisation on the host side - to ensure we only reserve the resources we need. And on the SmartNIC, where the resources are more liberally available, we protect the whole shared memory region once, at the start of the TAS process.

As part of using RDMA to communicate between the application and the fast and slow paths, we move away from a socket based communication mechanism. In TAS, sockets are used for notification between the application and the slow path. It is also through these sockets that address information for each of the RX, TX, and context queues per in the shared memory for each application is communicated. By using RDMA to read and write from the shared memory region on the NIC for our implementation, we take advantage of RDMA's ability to also wake up the relevant threads so they can process the data on the queues as necessary.

3 Implementation

We implement a prototype of the above design, the code for which is available on Github [1]. For the purpose of analysis, the implementation can be broadly segregated into three sections - `libTAS`, `tas_host`

[1] `https://github.com/ishank-arora/tas-smartnic`

and tas_nic. libTAS interfaces with the application, intercepting the linux system calls and running them on the TAS stack. tas_host implements the interface to the NIC and manages the RDMA protected region on the host side. tas_nic houses the fast and slow paths of TAS and manage the queues on the NIC side.

We make an important observation when implementing the design as mentioned in Figure 2. All the communication between the host and NIC can be abstracted into two categories: (1) queues, and (2) file descriptors. The queues are used per-application to hold the payload buffers and TX/RX contexts. And the file descriptors are used to wake up application and the fast path threads by each other.

For the queues, we implement a structure called the rdma_queue (see Figure 2) and a thin API in tas_host and a compatible API in tas_nic. The role of the API is to sync rdma_queues across the host and the NIC using a flush() function call. TAS originally used shared file descriptors to wake up application and fast path threads - when we move the fast and slow paths to the SmartNIC we needed an easy mechanism for this communication. To this end, we implement additional RDMA message types (RDMA_MSG_WAKEUP_TAS and RDMA_MSG_WAKEUP_APP) that notify the relevant thread on the other end of the RDMA connection. Finally, we replace all shared memory accesses between libTAS and TAS slow/fast path with calls to read/write and flush rdma_queues. And insert rdma messages at all places where the writes to file descriptors were used to wake up either application or fast path threads.

3.1 Implementation Challenges

Given the heavy engineering this project required, we would like to mention a few challenges we faced in implementing the proposed system design (figure 2).

Firstly, we came across an issue with the installation of DPDK and binding the vfio driver to it. The Bluefield SmartNIC does has no IOMMU protection available. It is important that to note that to use a VFIO-PCI driver successfully with DPDK on the SmartNIC, we need to turn on VFIO's no IOMMU mode. Or, alternatively, realise that the vendor Mellanox driver (mlx5_core) is compatible with DPDK. Even after figuring this out, we were unable to get the TAS process running. After some heavy debugging leading to nowhere, it took a fresh installation of DPDK on the SmartNIC that enabled us to continue.

Secondly, we encountered an issue with the inline assembly of the spinlock. We employed an atomic instruction instead, and that enabled us to finally get an echo server running between the SmartNIC and the host, as well as the SmartNIC and another machine on the network (dog1-lom). We're not sure why exactly we encountered this error, especially as we were using the code from the previous group that did successfully manage to get TAS running on the SmartNIC.

Engaging with the TAS codebase while still learning about the field of networking in general was quite challenging. It took us a long time to be able to connect the ideas of the TAS paper to the actual files and the details of their interactions. Many hours went into tracing how, and specifically where, each component of TAS interacted with each other - between the application, fast path, and the slow path to be able to make the necessary changes to shared memory. Furthermore, the asynchronous nature of libTAS, slow path and fast path resulted in race conditions among the processes, which were hard to debug.

Another issue we encountered was the lack of documentation and clear guidance on working with RDMA. This issue impacted us on both the implementation and debugging. A big part of this project went into understanding RDMA and IB verbs, along with understanding the insights from the paper FaRM: Fast Remote Memory (by Dragojević et al.) [2].

Lastly, while testing our implementation, we noticed that TAS kept dropping ARP request packets that the host sent to the NIC. After investigation, we learned that this was due to the ARP cache on the SmartNIC not being filled. TAS fails when this happens. To get around this issue, we create a static entry to prevent TAS from dropping the ARP request packets.

4 Evaluation

Due to the nature of this project, we focus heavily on the engineering aspect and delivering a prototype. We also realise that there is scope for improving and optimizing the implementation to achieve better performance. We do deliver a functioning prototype that is able to send and receive packets, albeit only some-

times. Our most important contribution is designing a system that is able to incorporate RDMA such that the fast and slow paths can be run on the SmartNIC, while the application runs on the host. In the rest of this section, we describe the limitations of our current prototype (Section 4.1) and the testing methodology we use for our prototype (Section 4.2).

4.1 Limitations

- Our prototype can only handle single segment messages correctly. We use a maximum segment size of 1448B, so as to replicate the common case scenario of datacenter RPCs, which are typically small.
- We observed that the `send()` calls may end up in a deadlock in our prototype. While we have not been able to pinpoint the exact reason for the same, we believe this error is orthogonal to the design choices we have made.
- Our prototype also eventually fails for connections that write more data than a certain threshold. We believe the reason for this is because of RDMA memory region overflow, but detailed profiling is needed.
- While our implementation on the host side only allocates a memory protected region for RDMA per application, all applications write and read from TAS_SHM (TAS shared memory) on the SmartNIC, which means that the TAS_SHM memory on SmartNIC shall be visible to an adversarial application as well. This is a security concern that should be addressed on the libTAS side by limiting the areas of TAS_SHM to which an application can make a RDMA read-/write call.

4.2 Testing Methodology

We test our prototype for two properties - (1) Correctness, and (2) Performance. To this end, we run an echo server application on the host, using TAS and connect to the server using a simple TCP client on a separate host. An echo server takes in a string as input from the client and then writes the same string back to the client.

Correctness: In our tests, we observe that the echo server receives the entire string correctly. However, we notice that our implementation is only correct for strings smaller than 1KB, as larger strings lead to the string being divided into multiple segments and our prototype fails to handle multiple segments in the RX queue correctly.

Performance: To evaluate the performance, we run the echo server benchmark for TAS and for vanilla TCP and compare on two parameters of latency and host CPU usage. Since our prototype can only handle small messages for short runs, it was not possible to run a heavy benchmark to compare the two. Hence, we use a light workload, where the client sends 100B messages to the server after every 100ms of wait period.

To evaluate the server-side latencies, we measure the average time for the `read()` call. We notice that while vanilla TCP took 30 μs per request on average, BlueTAS took 520 μs per request on average. Additionally, we observe that the idle CPU time during the testing interval was 48.27% for vanilla TCP and 48.62% for BlueTAS. We note that the slight improvement in the CPU idle times (or equivalently, CPU usage) might not be significant compared to the hit BlueTAS takes in latency.

It is noteworthy to mention that the RDMA `flush()` operation, which one might think to be the most critical operation, takes on an average 7 μs. This implies that the inefficiency of our prototype is not in the `flush()` operations, but elsewhere. We leave the detailed profiling of this as future work.

5 Future Work & Discussion

Immediately following this work, future work needs to address the limitations we have laid out.

First and foremost, it is important that the overheads of BlueTAS implementation are formally studied. From our experiments, we notice that the RDMA queue `flush()` operations do not impose a significant overhead. Therefore, the degradation in performance is due to some other inefficiency in the prototype, which needs to be studied carefully.

Similarly, more formal work is also required to compare the design we describe in this paper to an alternative design: where the application and the slow path run on the host, and only the fast path is offloaded to the SmartNIC. We suspect that it is better to have the fast path and slow path together on the SmartNIC, but it is only a suspicion, and thus should be formally addressed.

References

[1] T. Cui, W. Zhang, K. Zhang, and A. Krishnamurthy. *Offloading Load Balancers onto SmartNICs*, page 56–62. Association for Computing Machinery, New York, NY, USA, 2021.

[2] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson. Farm: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, Seattle, WA, Apr. 2014. USENIX Association.

[3] E. Jeong, S. Wood, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mtcp: a highly scalable user-level TCP stack for multicore systems. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 489–502, Seattle, WA, Apr. 2014. USENIX Association.

[4] A. Kaufmann, T. Stamler, S. Peter, N. K. Sharma, A. Krishnamurthy, and T. Anderson. Tas: Tcp acceleration as an os service. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[5] M. Liu, T. Cui, H. Schuh, A. Krishnamurthy, S. Peter, and K. Gupta. Offloading distributed applications onto smartnics using ipipe. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 318–333, New York, NY, USA, 2019. Association for Computing Machinery.

[6] I. Marinos, R. N. Watson, and M. Handley. Network stack specialization for performance. *SIGCOMM Comput. Commun. Rev.*, 44(4):175–186, aug 2014.

[7] Y. Qiu, J. Xing, K.-F. Hsu, Q. Kang, M. Liu, S. Narayana, and A. Chen. *Automated SmartNIC Offloading Insights for Network Functions*, page 772–787. Association for Computing Machinery, New York, NY, USA, 2021.