MIT 6.5810 Final Project: Adding RDMA to AIFM

Qing Feng

1 Introduction

Memory is typically the most contended and least elastic resource in in modern data centers. Works can be unnecessarily killed due to local memory depletion while on another server memory may be readily available. To address this problem, various far-memory systems are presented. *Application-integrated Far Memory* (AIFM) [9] is one of such state-of-the-art approaches, which makes remote memory available to local applications with an object-granular semantics and high performance.

A key insight for AIFM's high performance, different from other far-memory systems such as Fastswap [1] and Infiniswap [4], is the abstraction of application-level remote-able objects instead of typical virtual memory pages. As a result, its runtime can swap, prefetch and evacuate remote-able data structures intelligently and efficiently. Also, productive work can be done in the wait time leveraging the low-cost user-level thread context switch, whereas swapping virtual memory pages through the Linux kernel requires amplifying transferred data to the page size on small objects, and the wait time is wasted spinning when the kernel handles page faults.

However, AIFM currently uses TCP/IP connections for accessing remote data. Despite its highly optimized DPDK-based TCP/IP stack built on top of Shenango [7], AIFM's performance can still be improved by switching to a faster alternative such as *Remote Direct Memory Access* (RDMA), which are widely used in other high-performance far memory systems. The key advantage of RDMA compared with TCP/IP is that it supports zero-copy networking by enabling the network adapter to transfer data from the wire directly to application memory or vice versa, so that network communications bypass the kernel, leading to lower CPU overheads and thus lower communication latency.

The goal of this project is to implement a data transfer backend supporting RDMA for AIFM and evaluate the performance gain. With a RDMA backend, AIFM's far memory accesses are expected to have lower CPU overheads as now they do not need to enter the remote node's kernel.

2 RDMA Backend Design

RDMA architecture. The Virtual Interface Architecture (VIA) is a dominant model for commodity RDMA which can be achieved either with specialized InfiniBand network [8], or over Ethernet (RoCE) [5], among other alternatives like iWARP. They provide identical software RDMA APIs [10] (Verbs) despite using different hardware stacks underneath. For simplicity and consistency with previous works like Fastswap, we assume a network setup with InfiniBand.

RDMA APIs. In InfiniBand networks, end nodes communicate through an interface called queue pairs (QPs), each consisting of a send queue and a receive queue, plus a completion queue (CQ) associated with the QP. Applications access QPs by posting Verbs to the queues. Verbs can be one-sided (READ, WRITE, ATOMIC), which bypass the remote CPU, or two-sided (SEND/RECV), which involve CPU of both the sender and the receiver. Both categories of Verbs can be used to build high performance far-memory systems, such as FaRM [2] and FaSST [6].

For our RDMA backend in AIFM, one-sided READ-/WRITE should be desired as no remote CPU overheads are incurred, while two-sided APIs may be useful when some computation needs to be done on the server side, such as memory copy, which is now handled by the *remote agent* in AIFM. In the scope of our project, we choose a connection-based one-sided RDMA backend and disable the remote agent in AIFM. However, a combination of the two may be ideal to fully support AIFM and can be engineered in the future. We plan to start with a per-connection QP implementation, while a per-core QP design is better but involves modification to Shenango.

Far Memory Management. Currently, AIFM has the server (remote node) manage remote-able objects in far memory and the TCP/IP APIs transfer objects based on their data structure ID and object ID. However, one-sided RDMA APIs directly operate on specified far-memory regions. Therefore, we manage remote-able objects on the client side, and shift the APIs to be memory-region-based. Upon connecting to a remote node, the server allocates a contiguous memory chunk and shares the metadata with the client, which can be used to store remote-able objects as instructed by the one-sided RDMA READ/WRITE requests. In the project we will only focus on raw-pointer objects of which the object ID and size can be derived from the remote-able pointers, whereas more sophisticated data structures like hash tables can be complicated and require two-sided RDMA.

Polling Completion. When the RDMA device completes an operation, it creates a corresponding work completion entry into the associated completion queue so the application is notified. Ideally, productive work should be done during the polling intervals to reduce local CPU overheads. We designed three stages of polling completion mechanism where each stage is an optimized alternative of the former. First, we can have the RDMA thread to conduct a busy-poll as a straw-man version. In this case, the local CPU overhead does not decrease as much, despite elimination of server-side CPU involvement. Second, we can make the AIFM runtime to busy-poll the completion queue, so that multiple RDMA threads can rely on a shared spinning thread, and thus should reduce local CPU overheads. Third, the underneath scheduler, Shenango, currently has a dedicated core handling AIFM's TCP/IP stack, so ideally, we should modify Shenango to enable the dedicated core polling CQs, so that RDMA threads can sleep and be woke when the operation is completed, and at the same time productive work can be done by the application threads. In the project's scope, we plan to start with the first design, which minimize the modification to the AIFM runtime and the underneath scheduler.

3 Implementation

Straw-man RDMA Backend. We first build a stand-alone RDMA backend that supports one-sided READ/WRITE across multiple connections between a single pair of server and client (Fig.1). Since RDMA communication requires connections be established first, we use the RDMA connection manager (CM) library *librdmacm* to build connections through a series of CM events. In the context of RDMA, server refers to far memory node while client represents local machine, as one-sided RDMA is highly client-driven. An RDMA connection is essentially a QP with an associated CQ, wrapped in a struct *rdma_cm_id* with other user-specified contexts.

The server provides APIs to start and end a global RDMA server, of which at any time there should not be more than one instance. The server simply listens on a default port, waiting for a given number of connections. Upon the first connect request, the server allocates a 16GB buffer and a protected domain, registers a memory region (MR) accordingly, and returns the base address and remote key to the client. The server

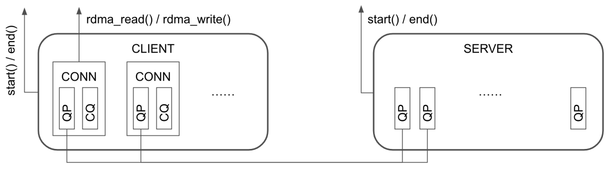

Figure 1: Our straw-man RDMA backend implementation

leverages the *private_data* field in the CM event parameters between events*RDMA_CM_EVENT_CONNECT_REQUEST* and *RDMA_CM_EVENT_ESTABLISHED* to send its MR metadata. By doing so, the server never allocates a completion channel nor a CQ as now the server does not actively send anything through RDMA, although there are also implementations such as Fastswap that uses two-sided SEND/RECV to exchange MR metadata.

The client provides APIs for one-sided RDMA read and write, besides starting and destroying a client instance. The client builds connections by calling a sequence of CM methods *rdma_resolve_addr*, *rdma_resolve_route* and *rdma_connect*, and storing the returned server metadata in the client instance. For a straw-man implementation, we make the client register a new MR for the provided buffer to work as the scatter-gather entry (SGE), so that data is directly read from and write into the given location. To process a read/write operation, the client can then build a work request (WR), post it to the send queue of the connection, busy poll the corresponding CQ by *ibv_poll_cq* until a CQ event returns, check the WR status and acknowledge the event. The MR is de-registered before the operation finishes.

Integration into AIFM. Now that we have a stand-alone RDMA backend, we plug it to the AIFM swapping code paths with minimal modification to AIFM.

As our RDMA backend only supports one-sided read and write, it is desired to preserve two-sided communication APIs needed for some AIFM functionality such as its *remote agent*, by making our *RDMADevice* class inherit the current *TCPDevice* and thus keeping the TCP connections intact. Upon construction, an *RDMADevice* instance starts a RDMA client and stores connections in a shared pool to be later used for WRs. To ensure there are always connections available, we set the number of RDMA connections as the product of the AIFM shared pool per-core cache size, and the most possible number of threads running concurrently on the machine, since RDMA connections are used busy polling so cannot be accessed by more than one thread at any time. The new *RDMADevice* class only overrides the *read_object* and *write_object* APIs, and only conducts RDMA operations when the given data structure ID refers to a raw-pointer object, for which we can derive the address and length locally. For writes, the object data length is provided, while for reads, we need to have the far memory manager read the data length from its metadata and pass it down to the RDMA APIs.

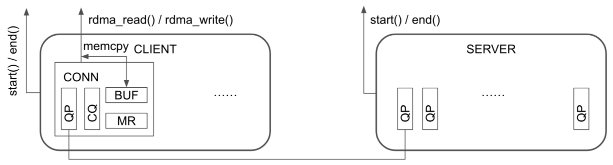

Figure 2: Our RDMA backend with pre-registered MR

On the remote node, the server spawns a separate thread to start an RDMA server and listen, besides spinning up the regular TCP stack. The RDMA server instance is destroyed altogether when the server processes shutdown through TCP. Otherwise, no extra CPU overheads are introduced.

Move MR off the Data Path. Benchmarks show that registering and de-registering new MRs are expensive and significantly slow down RDMA operations. We take a rather intuitive approach to address this issue by allocating a buffer and registering a reusable MR accordingly upon each RDMA connection construction on the client side. The architecture we implemented can be illustrated by Fig.2.

Upon operations, the client conducts one memory copy locally to shift data between the provided memory and the MR buffer in order to reuse the MR and thus avoid redundant registrations. Since the data length is bounded by the relatively small object size (64KB), the cost of this memory copy is negligible. One advantage of this implementation is that the RDMA backend remains query-agnostic, meaning no memory knowledge of upper level AIFM data structures is assumed. An alternative approach is that upon construction of the RDMA backend, it registers a large MR and passes the metadata up to AIFM memory manager so that raw-pointer objects are allocated in the specified region, but this would require more coupling with AIFM.

4 Evaluation

Setup. Since the goal is to measure the performance gain of switching to RDMA, we essentially replicate the experiments setup described in the AIFM paper [9] to yield comparable results, with two *xl170* nodes on CloudLab [3]. Notably, the RDMA backend needs to run on a different port from AIFM to avoid conflict, and we need to ensure AIFM and the RDMA backend run on the same 25Gbits/s Mellanox ConnectX-4 NIC for benchmark compatibility.

Benchmarks. Currently, the AIFM evaluation does not reflect a fair comparison between AIFM and Fastswap due to the different I/O stacks they rely on. We rerun the two benchmarks on cold path (remote object) latency and throughput as presented in Fig.12b in the AIFM paper, with object size set as 64B and 4KB. The results can be effectively compared with Fastswap and AIFM with TCP/IP, to demonstrate the performance improvement brought by our RDMA backend.

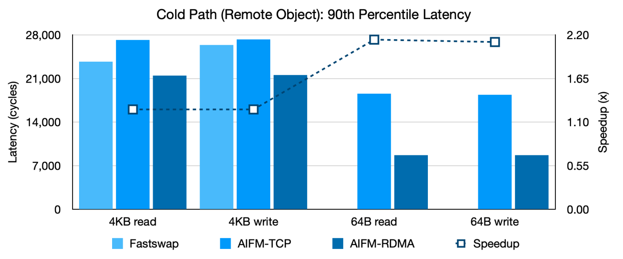

Figure 3: Remote objects swapping latency benchmarks

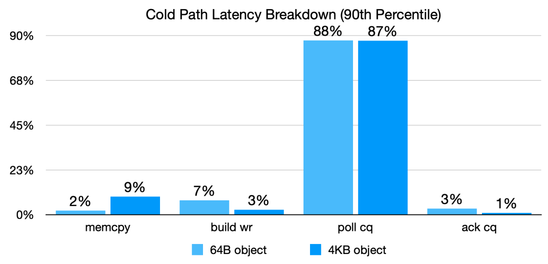

Figure 4: RDMA write runtime breakdown

The results are shown in Fig.3. Our benchmarks show AIFM with our RDMA backend demonstrates up to 2.1x speedup on 64B objects and approximately 1.25x speedup on 4KB objects, compared with the TCP alternative. In addition, our RDMA-backed AIFM outperforms Fastswap even on 4KB object raw data swapping, confirming that AIFM delivers not only lower overheads but also a better overall performance than Fastswap.

Our latency breakdown (Fig.4) shows that the RDMA operation latency is dominated by busy-polling. To further optimize our RDMA backend, better polling mechanism should be experimented, where ideally productive work should be done during CPU cycles waiting on CQ events to return.

We also experimentally benchmarked the AIFM cold path throughput with TCP and RDMA. The results show that the two backends perform very similarly in terms of throughput, possibly both are already capped by network physical boundary.

End-To-End Performance. As our RDMA backend implementation preserves the TCP connections to fully support AIFM functionalities, it passes all AIFM tests successfully, despite some known bugs during RDMA client cleaning up. To demonstrate the end-to-end performance gain, we managed to rerun the the end-to-end experiments on the Snappy library as described in the AIFM evaluation section 8.2.2, where solely raw-pointer objects (arrays) are used. The results show that switching to our RDMA backend brings very slight speedup (approx. 0.2%), as the original AIFM already performs very close to optimal (all in local memory). However, as server side CPU overheads are removed, our RDMA backend uses fewer CPU cores compared with the TCP counterpart.

We should also be able to rerun all the end-to-end experiments as described in the AIFM paper. However, the performance gain of our RDMA backend is best illustrated in the above benchmarks where raw-pointer object swapping is focused while end-to-end benchmarks require significant further engineering efforts. As a result, for the scope of this project, we leave most end-to-end benchmarks for future works. Speculatively, as of now the performance gain would be less visible since many AIFM data transfers are still using TCP/IP.

5 Conclusion

In this project, we implemented an RDMA backend that can be used in AIFM for more efficient data transfer between local and remote nodes. Our design leverages one-sided RDMA to eliminate CPU involvement on remote machine, providing lower overheads as well as better performance isolation. The implementation minimizes modifications to AIFM and is highly portable to other TCP-based systems, delivering lower CPU overheads on data transfer tasks.

Our evaluation shows that our RDMA-backed AIFM leads to approximately 2.1x speedup on 64B objects and 1.25x speedup on 4KB objects during remote object swapping, compared with the highly optimized TCP/IP stack. Our work also confirms that AIFM outperforms Fastswap both in terms of non-data-transfer overheads and overall performance, on both small and page-sized objects. By switching from TCP to RDMA, we eliminate server side CPU overheads and essentially save a good portion of cores needed in the system.

If we were to further work on this project in the future, we would like to optimize our implementation of polling completion, as described in section 2, given that currently RDMA runtime is dominated by busy-polling the CQ. In addition, there are AIFM features that require two-sided RDMA such as fancier remot-able data structures like hash tables. We would also like to implement two-sided RDMA operations to completely replace TCP in AIFM. Ideally, with the two mentioned optimizations implemented, we should be able to comprehensively evaluate the end-to-end performance improvement of AIFM.

6 Acknowledgements

We thank Adam Belay and Zhenyuan Ruan for instructions over the course and help with AIFM, and Cloudlab [3] for the testbed. Our implementation references code from Fastswap [1] and example from Animesh Trivedi. Our code can be found at https://github.com/fengqingthu/MIT_6.5810_Adding_RDMA_To_AIFM.

References

[1] E. Amaro, C. Branner-Augmon, Z. Luo, A. Ousterhout, M. K. Aguilera, A. Panda, S. Ratnasamy, S. Shenker. Can far memory improve job throughput? *Proceedings of the Fifteenth European Conference on Computer Systems*, 1–16, 2020.

[2] A. Dragojević, D. Narayanan, M. Castro, O. Hodson. {FaRM}: Fast remote memory. *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, 401–414, 2014.

[3] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, *et al.* The design and operation of {CloudLab}. *2019 USENIX annual technical conference (USENIX ATC 19)*, 1–14, 2019.

[4] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, K. G. Shin. Efficient memory disaggregation with infiniswap. *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 649–667, 2017.

[5] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, M. Lipshteyn. Rdma over commodity ethernet at scale. *Proceedings of the 2016 ACM SIGCOMM Conference*, 202–215, 2016.

[6] A. Kalia, M. Kaminsky, D. G. Andersen. FaSST: Fast, scalable and simple distributed transactions with Two-Sided (RDMA) datagram RPCs. *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 185–201. USENIX Association, Savannah, GA, 2016. ISBN 978-1-931971-33-1.

[7] A. Ousterhout, J. Fried, J. Behrens, A. Belay, H. Balakrishnan. Shenango: Achieving high {CPU} efficiency for latency-sensitive datacenter workloads. *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 361–378, 2019.

[8] G. F. Pfister. An introduction to the infiniband architecture. *High performance mass storage and parallel I/O*, **42**(617-632), 102, 2001.

[9] Z. Ruan, M. Schwarzkopf, M. K. Aguilera, A. Belay. {AIFM}:{High-Performance},{Application-Integrated} far memory. *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 315–332, 2020.

[10] B. Woodruff, S. Hefty, R. Dreier, H. Rosenstock. Introduction to the infiniband core software. *Linux symposium*, vol. 2, 271–282, 2005.