Dagger: Towards Efficient RPCs in Cloud Microservices with Near-Memory Reconfigurable NICs

Nikita Lazarev, Neil Adit, Shaojie Xiang, Zhiru Zhang, and Christina Delimitrou

Abstract— Cloud applications are increasingly relying on hundreds of loosely-coupled microservices to complete user requests that meet an application’s end-to-end QoS requirements. Communication time between services accounts for a large fraction of the end-to-end latency and can introduce performance unpredictability and QoS violations. In this work, we present our early work on Dagger, a hardware acceleration platform for networking, designed specifically with the unique qualities of microservices in mind. The architecture relies on an FPGA-based NIC, closely coupled with the processor over a configurable memory interconnect, designed to offload and accelerate RPC stacks. Unlike the traditional cloud systems with PCIe links as the de facto NIC I/O interface, which lack the efficiency needed for fine-grained microservices and add non-negligible overheads, we leverage memory-interconnected FPGAs as networking devices to make I/Os more efficient, transparent and programmable. We show that this considerably improves CPU utilization, performance, and RPC scalability in cloud microservices.

Index Terms— Microservices, programmable NICs, RPC, memory interconnects, FPGA, near-memory processing.

1 INTRODUCTION

The past few years have seen a major shift in the way cloud applications are designed, from traditional monolithic architectures to microservices. Large applications are split into many loosely-coupled and single-purpose components communicating with each other over the network. While microservices have several advantages, including improved development and deployment cycles, and error isolation, they also introduce non-negligible system overheads. Since individual microservices are typically not overly computationally intensive [10], [24], this overhead is mostly introduced in the networking stack, with the latter gaining increased attention both by application and system designers.

Remote Procedure Calls (RPC) is one of the most common communication techniques in microservices. There is a variety of available RPC frameworks, however, since these frameworks were not designed with microservices specifically in mind, they do not address their unique resource requirements. Unlike traditional monolithic and coarse-grained distributed applications, microservices exhibit different traffic patterns and performance requirements. Thus they demand a fundamental reconsideration of design and architectures of networking systems.

As the demand for high-bandwidth and low-latency networking in the cloud continues to grow, research from both industry and academia has offered numerous proposals that approach the problem at different levels of the networking stack. Some of these proposals optimize transport protocols [3], [12], [17] for low latency networking, while others leverage hardware-assisted system design solutions, such as user-space networking [5], [13], RDMA [8], [15], reconfigurable FPGA NICs [6], [21], and multicore smartNICs [7], [16]. While these works demonstrate the potential of hardware-software co-design to improve the performance and efficiency of cloud networking, these systems are designed for traditional monolithic applications, and are all based on PCIe-attached NICs, so they inherit the I/O inefficiencies coming from this de facto CPU-NIC interconnection method [18]. Moreover, most of the aforementioned proposals require laborious software engineering to work on a given NIC, which is especially challenging given that most commercial NIC implementations are closed source [14].

In this work, we focus on a hardware-accelerated platform for RPCs in the context of interactive microservices. We observe that today’s PCIe-based CPU-NIC interconnects lack the efficiency required for microservices with µs-scale performance and are limited to MMIO+DMA/DDIO data transferring modes, which introduce significant overheads when dealing with small RPC requests under strict latency and throughput requirements. Instead, we propose to leverage memory interconnects (NUMA) as the interface between CPUs and NIC, where the latter accommodates the RPC stack and is integrated into the processor’s memory sub-system. Compared to PCIe interfaces, memory intercon-

* All authors are with the Department of Electrical and Computer Engineering, Cornell University, Ithaca, NY, 14853.
E-mail: {ns524, na4699, sx233, zhiru, delimitrou}@cornell.edu
Manuscript received June XXX, 2020; revised August XXX, 2020.

Microservices have distinct network requirements and traffic compared to monolithic applications and traditional distributed systems. First, every user request in microservices is propagated through a large graph of tiers, with per-node processing and communication delays being accumulated to the end-to-end latency. As a result, the Quality of Service (QoS) which is usually defined in terms of tail latency, under a certain load (Queries per Second (QPS)) critically depends on the performance of every communication channel between each pair of microservices on the call path. Hence, even a small latency increase in the networking stack translates to considerable increases in end-to-end latency, as shown in Figure 1 which plots the end-to-end fractions of networking and application (including queuing) time w.r.t the load.

Second, even though RPC request and response payloads in typical datacenter applications are already relatively small, ranging from hundreds of bytes to few kBytes [11], [17], [28], in microservices that number is even smaller, as shown in Figure 2 for the Social Network and Media Service from DeathStarBench [10].

Network Characteristics in Microservices

Microservices have distinct network requirements and traffic compared to monolithic applications and traditional distributed systems. First, every user request in microservices is propagated through a large graph of tiers, with per-node processing and communication delays being accumulated to the end-to-end latency. As a result, the Quality of Service (QoS) which is usually defined in terms of tail latency, under a certain load (Queries per Second (QPS)) critically depends on the performance of every communication channel between each pair of microservices on the call path. Hence, even a small latency increase in the networking stack translates to considerable increases in end-to-end latency, as shown in Figure 1 which plots the end-to-end fractions of networking and application (including queuing) time w.r.t the load.

Second, even though RPC request and response payloads in typical datacenter applications are already relatively small, ranging from hundreds of bytes to few kBytes [11], [17], [28], in microservices that number is even smaller, as shown in Figure 2 for the Social Network and Media Service from DeathStarBench [10].
As seen from Figure 2, more than 70% of RPC requests are smaller than 256B and almost all requests are within 1280B. Responses are even smaller: nearly 100% of messages are less than 256B with 95% of messages fitting 64B. These tiny messages introduce high pressure on networking stacks at all levels, and previous work has shown that commodity networking systems cannot efficiently handle this traffic, due to high per-packet overheads [5], [13], [17].

Finally, microservices are by design very diverse in terms of design patterns and performance requirements [23]. In particular, there is a rich variety of thread models [25], network queuing architectures [27], and different strategies of mapping microservices to the available hardware resources. Performance requirements also vary a lot, with some microservices being latency-critical while others are treated as batch. Commodity networking systems were initially designed with generality and compatibility in mind, in order to fit any request features and, therefore, do not necessarily provide the most efficient solution for a particular application class. This has caused programmable networking systems to become more popular [4], [20]. Such systems allow flexibly adjusting networking primitives depending on the currently-running applications. Even so, one part of the networking stack that is always fixed in silicon, even in programmable network controllers, is the NIC I/O interface.

3 DAGGER: A NEAR-MEMORY RECONFIGURABLE NIC FOR INTERACTIVE MICROSERVICES

In this work, we propose to leverage FPGAs closely coupled with server-class CPUs as reconfigurable RPC acceleration fabrics, which are designed with the unique properties and requirements of interactive microservices in mind. Such FPGA-enabled platforms are already available in commercial server systems like Intel® Xeon®.

3.1 Motivation for Near-Memory NICs

PCIe is the current de facto standard interface between CPU and NICs or accelerators. Unfortunately, the PCIe bus has limited functionality, it requires multiple bus transactions and memory synchronization primitives require sending data chunks to the device which makes per-packet overheads large [14]. The regular way to send data over PCIe to a peripheral is by using DMA transfers together with expensive initiation transactions explicitly issued by the processor as MMIO requests. This MMIO+DMA combination, alongside with mandatory synchronization instructions, promotes strong memory consistency over achieving high performance for fine-grained data transfers.

The ideal CPU-NIC interconnect for modular interactive services should avoid any CPU-initiated requests and explicit memory synchronization, and should track data transfers entirely in hardware buffers without the processor’s intervention. In addition, it can sacrifice strong consistency for a weaker memory consistency model in favor of performance, since interactive services have high request-level parallelism and are tolerant to message ordering. Most commercially-available memory interconnects satisfy these requirements: they provide efficient relaxed memory consistency and implement cache coherency state machines that can track updates in networking buffers without CPU intervention.

3.2 The Need for Reconfigurable and Transparent NICs

As shown in Section 2, microservices are very diverse in terms of traffic patterns and performance requirements. Prior work has also established that specially optimized transports are required to get high throughput [12], and that such protocols usually do not work well for latency-critical applications [5], [17], highlighting the need for reconfigurable transport layers [4].

We argue that the same holds for CPU-NIC interfaces. For example, the standard MMIO+DMA mode works well when transferring large packets, however, it performs poorly when it comes to delivering a large number of small requests. The consistency model requirements can also vary across applications: for requests that fit into the interconnect MTU and for the applications that do not require strict RPC ordering, a weak memory consistency model will achieve the highest performance. Finally, software design patterns also differ: some systems provision network buffers on a per-connection basis [8], [15], while others use shared or single-queue solutions to handle load balancing and improve scalability [27]. Despite this variability, the hardware support on the NIC is required to efficiently run whichever queue provisioning schemes [26].

This variability introduces the potential to tailor the acceleration fabric and CPU-FPGA interface to the requirements of a given application model. At the same time, it introduces no-trivial complexity, which users should not have to manage themselves. In light of the above considerations, we present our early work on a reconfigurable and transparent-to-the-user hardware acceleration fabric for microservice RPCs, with a tunable CPU-NIC interface which can be easily altered to fit the requirements of a given application.

3.3 Dagger Platform Overview

We implement a prototype of our FPGA NIC on the Intel® Broadwell® integrated CPU/FPGA architecture with different programmable options for the CPU-NIC interface over CCI-P [1] bus based on both commodity PCIe and coherent memory (UPI) interconnects. CCI-P is selected because of the following four features: (1) Relaxed memory consistency that can enable much faster transfer of small RPC requests in settings where per-packet ordering is not critical. (2) The ability to strengthen consistency models (up to sequential) on-the-fly, when required by the application. (3) The possibility to monitor invalidation transactions and use them to initiate data transfers entirely in hardware, instead of the slow software-issued MMIOs. (4) The flexible choice of the low-level physical and link layers of the interconnect between PCIe and UPI. We use CCI-P to communicate ready-to-use RPC objects with the processor’s LLC. Figure 3 shows the top-level overview of our prototype.
4 Preliminary Results

We built Dagger using an Intel Broadwell CPU/FPGA chip (Xeon® E5-2600, 2.3GHz), available in the HARP academic research platform. Due to hardware limitation of the platform, we are only presenting a subset of the CCI-P schemes described in Section 3.3. In our experiments, we run a simple concurrent P2P client-server application sending 64B echo RPCs.

4.1 Single Request Latency

Figure 5 shows the round trip time of 64B synchronous (blocking) RPC requests with different CCI-P messaging schemes.

Table 1: Round trip times of synchronous RPCs vs. related work.

| Objects | IX | eRPC (CX4) | eRPC (CX5) | NetDIMM | Dagger |
|---------|----|------------|------------|---------|--------|
| TOR delay | N/A | 0.3 us | 0.22 us | 0.1 us | 0.1 us |
| RTT      | 11.4 us | 3.7 us | 2.3 us | 2.2 us | 1.93 us |

4.2 Throughput

As seen from Table 2, Dagger’s memory interconnect outperforms the DMA+MMIO transfer by 7 – 9×. We confirm this observation by running another microbenchmark where we write to a remote memory location using MMIO compared to UPI. Similarly, writing over UPI offers 10 – 14× better throughput. Table 2 also shows that CPU LLC polling (mode #1, Dimension A) achieves goodput of up to 18.4 Mrps.
which is $3 \times$ better than mode #2. Even though the polling mode may not be practical due to the high bus bandwidth consumption, it shows the potential of memory interconnects for hardware-accelerated networking. Moreover, since the CCI-P messaging scheme can be easily configured on the fly, the NIC can dynamically switch from mode #2, which is more energy efficient to #1 when the load increases.

Table 2 shows the RPC receiving rate of Dagger, which reaches 12.4Mps for $B = 4$. Since this is lower than the throughput of sending requests, the receiving path is the current system bottleneck. Nonetheless, Dagger noticeably outperforms previous solutions [5], [13], [15]. We also observe that in our current implementation, the goodput is limited by the software side of the networking stack, i.e., writing/reading TX and RX buffers. We plan to further optimize the software stack as part of our future work.

Table 2: Goodput of asynchronous transfer of 64B RPCs on a single core; B denotes CCI-P batching.

| B  | Sending rate | Recv. rate |
|----|--------------|------------|
|    | MMIO/DMA     | Mode #2    | Mode #1    |
| 1  | 0.43 Mrps    | 4.1 Mrps   | 11.8 Mrps  | 8.2 Mrps   |
| 2  | 0.61 Mrps    | 5.3 Mrps   | 14.3 Mrps  | 9.4 Mrps   |
| 4  | 0.83 Mrps    | 6.2 Mrps   | 18.4 Mrps  | 12.4 Mrps  |

Figure 6 (Left) shows the latency-goodput curves across loads. Since we run a simple echo microbenchmark here, the system immediately blocks the caller thread when goodput is saturated, shown with vertical dotted lines. The latency remains stable across the entire load range. The figure also shows how CCI-P batching might affect system performance: $B = 4$ allows high goodput, however, it increases the request latency under low QPS. Since Dagger can reconfigure the batch size on-the-fly, it can dynamically adjust the CCI-P batching depending on the current load, as shown by the green dashed line.

Figure 6 (Right) shows the goodput scalability of our system with the number of CPU cores. Dagger achieves 68Mps of the RPC issue rate by the client and 40Mps of the end-to-end client-server median goodput with 8 cores. This result is $3.8 \times$ better than the best RDMA-based solution, FASST [15], and $5.7 \times$ better than DPDK-based IX [5]. When using a more efficient CCI-P messaging scheme based on invalidation messages (mode #2, Dimension A in Section 3.3), Dagger still outperforms previous works with end-to-end median goodput of 25Mps on 8 cores.

5 Conclusion

In this paper, we present implementation of a prototype RPC system on a closely-coupled near-memory FPGA and show its performance advantages over existing RPC frameworks built on top of commodity PCIe-attached NICs, therefore demonstrating in practice why NICs should be integrated into the processor’s memory sub-system. The approach significantly increases RPC goodput while showing state-of-the-art round-trip latency. Our prototype outperforms existing user-space networking- and RDMA-based solutions based on peripheral networking devices and, in addition, provides transparent NIC I/O at the hardware level. Overall, we show that the closely-coupled CPUs and FPGAs can be used as efficient programmable networking devices that drastically improve networking I/O for fine-grained workloads.

References

[1] “Intel accelerator stack for intel xenon cpu with fpgas core interface (cci-p) reference manual,” accessed May, 2020, https://www.intel.com.
[2] M. Alian and N. S. Kim, “Netdimm: Low-latency near-memory network interface architecture,” in Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, 2019.
[3] M. Alizadeh, A. Javanmard et al., “Analysis of dcetcp: Stability, convergence, and fairness,” in Proceedings of the ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems, 2011.
[4] M. T. Arashilo, A. Lavrov et al., “Enabling programmable transport protocols in high-speed nics,” in 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20), 2020.
[5] A. Belay, G. Prekas et al., “IX: A protected dataplane operating system for high-speed AI/ML and low latency,” 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14).
[6] M. Blott, K. Karras et al., “Achieving 10gbps line-rate key-value stores with fpgas,” in Presented as part of the 5th USENIX Workshop on Hot Topics in Cloud Computing, 2013.
[7] A. Caulfield, P. Costa et al., “Beyond smartnets: Towards a fully programmable cloud: Invited paper,” 2018.
[8] A. Dragojević, D. Narayanan et al., “Farm: Fast remote memory,” in 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14), 2014.
[9] M. Flajslik and M. Rosenblum, “Network interface design for low latency on a closely-coupled near-memory FPGA and show its performance validation messages (mode #2, Dimension A in Section 3.3), Dagger still.
[10] Y. Gan, Y. Zhang et al., “Characterizing facebook’s memcached workloads,” in Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, 2016.
[11] Q. Huang, K. Birman et al., “An analysis of facebook photo caching,” in Proceedings of the Twentieth ACM Symposium on Operating Systems Principles, 2013.
[12] E. Jeong, S. Wood et al., “mtcp: a highly scalable user-level TCP stack for multicore systems,” in 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14), 2014.
[13] A. Kalia, M. Kaminsky et al., “Datacenter rpcs can be general and fast,” in 16th USENIX Symposium on Networked Systems Design and Implementation, 2019.
[14] A. Kalia, M. Kaminsky et al., “Design guidelines for high performance RDMA systems,” in 2016 USENIX Annual Technical Conference (USENIX ATC 16), 2016.
[15] A. Kalia, M. Kaminsky et al., “Fasst: Fast, scalable and simple distributed transactions with two-sided (rdma) datagram rpcs,” in Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, 2016.
[16] M. Liu, S. Peter et al., “E3: Energy-efficient microservices on smartnic-accelerated servers,” in Proceedings of the 2019 USENIX Conference on Userland Technical Conference, USA, 2019.
[17] B. Montazeri, Y. Li et al., “Homa: A receiver-driven low-latency transport protocol using network priorities,” in Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, 2018.
[18] R. Neugebauer, G. Antichi et al., “Understanding pec performance for end host networking,” in Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, 2018.
[19] S. Novakovic, A. Daglis et al., “Scale-out numa,” in Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems, 2014.
[20] P. M. Phothilimthana, M. Liu et al., “Floem: A programming system for nic-accelerated network applications,” in Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, 2018.
[21] A. Putnam, A. Caulfield et al., “A reconfigurable fabric for accelerating large-scale datacenter services,” in Proceedings of the 41st Annual International Symposium on Computer Architecture (ISCA), June 2014.
[22] D. Sidler, Z. Istvan et al., “Low-latency tcp/ip stack for data center applications,” in 2016 26th International Conference on Field Programmable Logic and Applications (FPL), 2016.
[23] A. Sriraman, A. Dhanotia et al., “Softsku: Optimizing server architectures for microservice diversity at scale,” in Proceedings of the 46th International Symposium on Computer Architecture, 2019.
[24] A. Sriraman and T. F. Wenisch, “yusite: A benchmark suite for microservices,” 2018.
[25] A. Sriraman and T. F. Wenisch, “pune: Auto-tuned threading for OLDDI microservices,” in 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), 2018.
[26] B. Stephens, A. Singhvi et al., “Titan: Fair packet scheduling for commodity multiqueue nics,” in 2017 USENIX Annual Technical Conference (USENIX ATC 17), 2017.
[27] M. Sutherland, S. Gupta et al., “The nebula rpc-optimized architecture,” [Proceedings of ISCA 2020].
[28] Y. Xu, E. Frachtenberg et al., “Characterizing facebook’s memcached workload,” IEEE Internet Computing, 2014.