# Dagger: Efficient and Fast RPCs in Cloud Microservices with Near-Memory Reconfigurable NICs

Nikita Lazarev
Cornell University
Ithaca, New York, USA
nl524@cornell.edu

Shaojie Xiang
Cornell University
Ithaca, New York, USA
sx233@cornell.edu

Neil Adit
Cornell University
Ithaca, New York, USA
na469@cornell.edu

Zhiru Zhang
Cornell University
Ithaca, New York, USA
zhiruz@cornell.edu

Christina Delimitrou
Cornell University
Ithaca, New York, USA
delimitrou@cornell.edu

## ABSTRACT

The ongoing shift of cloud services from monolithic designs to microservices creates high demand for efficient and high performance datacenter networking stacks, optimized for fine-grained workloads. Commodity networking systems based on software stacks and peripheral NICs introduce high overheads when it comes to delivering small messages.

We present *Dagger*, a hardware acceleration fabric for cloud RPCs based on FPGAs, where the accelerator is closely-coupled with the host processor over a configurable memory interconnect. The three key design principle of Dagger are: (1) offloading the entire RPC stack to an FPGA-based NIC, (2) leveraging memory interconnects instead of PCIe buses as the interface with the host CPU, and (3) making the acceleration fabric reconfigurable, so it can accommodate the diverse needs of microservices. We show that the combination of these principles significantly improves the efficiency and performance of cloud RPC systems while preserving their generality. Dagger achieves $1.3 - 3.8\times$ higher per-core RPC throughput compared to both highly-optimized software stacks, and systems using specialized RDMA adapters. It also scales up to 84 Mrps with 8 threads on 4 CPU cores, while maintaining state-of-the-art $\mu$s-scale tail latency. We also demonstrate that large third-party applications, like memcached and MICA KVS, can be easily ported on Dagger with minimal changes to their codebase, bringing their median and tail KVS access latency down to $2.8 - 3.5$ us and $5.4 - 7.8$ us, respectively. Finally, we show that Dagger is beneficial for multi-tier end-to-end microservices with different threading models by evaluating it using an 8-tier application implementing a flight check-in service.

## CCS CONCEPTS

• **Hardware → Networking hardware**; • **Networks → Cloud computing**; *Programmable networks.*

## KEYWORDS

End-host networking, cloud computing, datacenters, RPC frameworks, microservices, smartNICs, FPGAs, cache-coherent FPGAs

**ACM Reference Format:**
Nikita Lazarev, Shaojie Xiang, Neil Adit, Zhiru Zhang, and Christina Delimitrou. 2021. Dagger: Efficient and Fast RPCs in Cloud Microservices with Near-Memory Reconfigurable NICs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21), April 19–23, 2021, Virtual, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3445814.3446696

## 1 INTRODUCTION

Modern cloud applications are increasingly turning to the microservices programming model to improve their agility, elasticity, and modularity [2, 3, 16, 31–34, 55, 56, 60]. Microservices break complex monolithic applications, which implement the entire functionality as a single service, into many fine-grained and loosely-coupled tiers. This improves design modularity, error isolation, and facilitates development and deployment. However, since microservices communicate with each other over the network, they also introduce significant communication overheads [33, 54]. Given that individual microservices typically involve a small amount of computation, networking ends up being a large fraction of their overall latency [33, 55]. Furthermore, since microservices depend on each other, performance unpredictability due to network congestion can propagate across dependent tiers and degrade the end-to-end performance [17, 25, 34, 37, 43].

Microservices typically communicate with each other over Remote Procedure Calls (RPC) [5, 13, 15]. Unfortunately, existing RPC frameworks were not designed specifically for microservices, whose network requirements and traffic characteristics differ from traditional cloud applications, and therefore introduce significant overheads to their performance. The strict latency requirements, fine-grained requests, wide diversity, and frequent design cadence of microservices put a lot of pressure on the network system, and makes rethinking networking with microservices in mind a pressing need. Most of the existing commercial RPC frameworks are implemented on top of commodity OS networking, such as Linux TCP/IP. While this ensures generality, such systems suffer from considerable overheads across all levels of the system stack [23, 54]. These overheads accumulate over deep microservice call paths, and result in end-to-end QoS violations. While this affects all microservices, it is especially challenging for interactive tiers, which optimize for low tail latency, instead of average throughput.

The past decade has seen increased interest both from academia and industry for lower latency and higher throughput networking systems. One line of work focuses on optimizing transport protocols [19, 20, 36, 46], while another moves networking to user space [4, 23, 36, 38], or offloads it to specialized adapters [27, 35, 40, 45, 47]. Network programmability has also gained traction through the use of SmartNICs [24, 29, 44] to tune the network configuration to the performance requirements of target applications. Despite the performance and efficiency benefits of these approaches, they are limited in the type of interfaces they use between the host CPU and the NIC. Almost all commercially available NICs are viewed by the processor as PCIe-connected peripheral devices. Unfortunately, PCIe interconnects require multiple bus transactions, memory synchronizations, and expensive MMIO requests for every request to the NIC [18, 30, 39, 49]. As a result, the per-packet overhead in these optimized systems is still high; this is especially noticeable for fine-grained workloads with deep call paths, like microservices.

This paper presents *Dagger*, an FPGA-based reconfigurable RPC stack integrated into the processor's memory subsystem over a NUMA interconnect. Integrated and near-memory NICs have already shown promise in reducing the overheads of PCIe, and improving networking efficiency [18, 50, 59]. However, prior integrated NICs are based on ASICs that lack reconfigurability, and require taping out custom chips, which is expensive and time consuming for frequently-changing networking configurations at datacenter scale. Widely-used RPC stacks for microservices, such as Thrift RPC [15], gRPC [13], offer a rich variety of transport options, (de)serialization methods, and threading models. Hardware-based RPC stacks can only be practical in the context of microservices if they allow the same flexibility. To this end, we propose an *integrated* and *reconfigurable* FPGA-accelerated networking fabric, capable of supporting realistic and end-to-end RPC frameworks.

Dagger is based on three key design principles: (1) The NIC implements the entire RPC stack in hardware, while the software is only responsible for providing the RPC API. This way, we remove CPU-related overheads from the critical path of RPC flows, and free more CPU resources for the high concurrency of microservices. (2) Dagger leverages memory interconnects to communicate with the processor. We show that in contrast to PCIe protocols that were initially designed for the Producer-Consumer dataflow pattern, memory interconnects offer a better communication model that is especially beneficial for transferring ready-to-use RPC objects. We also argue that integrating NICs via memory interconnects is more practical than previously-proposed methods of closely-coupling NICs with CPUs, since processors today come with exposed memory busses, with the next generation of server-class CPUs already offering dedicated peripheral memory interconnects [12]. (3) Finally, Dagger is based on an FPGA, so its design is fully programmable. This allows it to adjust to the performance and resource requirements of a given microservice.

We build Dagger on an Intel Broadwell CPU/FPGA hybrid architecture, similar to those available in public clouds, such as Intel HARP. We show that offloading the entire RPC stack to hardware enables better CPU efficiency, which results in higher per-core RPC throughput and lower request latency. In addition, we demonstrate the benefits of closely-coupling hardware RPC stacks with applications through memory interconnects. Dagger improves the per-core RPC throughput up to 1.3-3.8× compared to prior work, based on both optimized software RPC frameworks [38] and specialized hardware adapters [40]. Dagger reaches 12.4 - 16.5 Mrps of per core throughput, and it scales up to 42 Mrps with only 4 physical threads on two CPU cores, while achieving state-of-the-art $\mu$s-scale end-to-end latency.

In addition, we show that Dagger can be easily integrated into existing datacenter applications with minor changes to the codebase. Our experiments with memcached and MICA KVS using Dagger as the communication layer show that it achieves median and $99^{th}$ percentile tail latency of 3.2 and 7.8 us respectively for memcached and 3.5 us and 5.7 us for MICA, while also achieving a throughput of 5.2 Mrps [6] on a single core. This result is 11.4× lower than the latency of memcached over a native transport based on the Linux kernel networking, and $4.4 - 5.2\times$ lower than of MICA over a highly-optimized, DPDK-based user space networking stack. Finally, we demonstrate that Dagger can accommodate multi-tier microservice applications with diverse requirements and threading models by porting an 8-tier Flight Registration service on top of Dagger, and showing significant performance benefits compared to native execution.

## 2 RELATED WORK

Designing low-latency networking systems including optimizations for small requests is not a new problem. Both industry and academia have contributed various proposals to this end. In this section, we briefly review prior work on networking acceleration, and discuss how it resembles and differs from our proposal. We classify related work into three categories: (1) software solutions, (2) systems leveraging specialized commercial hardware adapters, and (3) proposals of new hardware architectures for efficient networking.

**Software-based solutions:** At the software level, most research has focused on transport protocol optimizations for low latency and/or high throughput [19, 20, 36, 46, 48]. This includes optimizing the congestion control mechanisms, flow scheduling, connection management, etc. In addition to transport optimizations, some proposals also suggest moving networking from the kernel to user space by leveraging, for example, DPDK [11, 23, 36, 41, 46] or raw NIC driver APIs [38]. Although these proposals demonstrate their efficiency in improving the performance of datacenter networks, they are still subject to system overheads due to their software-only and CPU-based implementation.

**Specialized commercial adapters:** As an alternative to software/ algorithmic-only optimizations, another line of work proposed to leverage RDMA hardware to offload network processing to specialized adapters, and use the remote memory abstraction to implement higher-level communication primitives, such as RPCs [27, 40, 57]. This approach improves CPU efficiency, and as a consequence also incurs lower latency and higher throughput. Despite this, there are two main issues with existing RDMA-related work. First, prior work does not implement a fully-offloaded RPC protocol; commodity RDMA adapters only offload the networking part, i.e., up to the transport layer and RDMA protocols, keeping the execution of

RPC layers on the host CPU. Second, all RDMA NICs are seen as peripheral devices from the perspective of the host CPUs, and are interconnected with the latter over PCIe busses that have been shown to be inefficient, especially when the metric of interest is latency and network packets are small [30, 39, 49]. Therefore, bringing NICs closer to CPUs and/or memory is required to enable efficient and fast communication in datacenters.

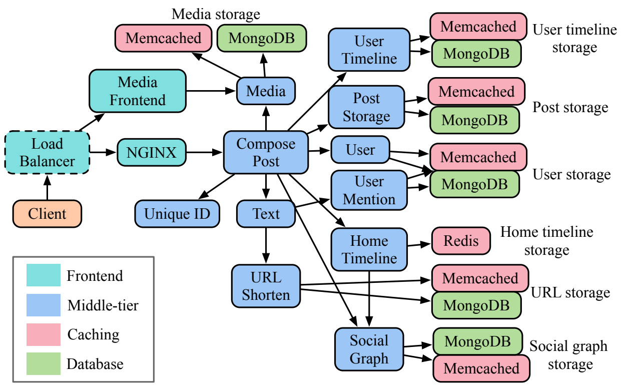


**Figure 1: Social Network microservice architecture [33]** – client requests first reach a front-end load balancer, which evenly distributes them across the $N$ webserver instances. Then, depending on the type of user request, mid-tiers will be invoked to create a post, read a user's timeline, follow/unfollow users, or receive recommendations on new users to follow. At the right-most of the figure, the requests reach the back-end databases, implemented both with in-memory caching tiers (memcached/Redis), and persistent databases (MongoDB).

**Integrated special-purpose networking hardware:** To address the aforementioned issues, a recent line of research proposed to tightly integrate NICs with the processor and memory. In particular, NetDIMM [18] investigates the potential of physical integration of NICs into DIMM memory hardware. However, NetDIMM does not focus on RPC stacks, and it requires designing custom ASICs inside the DIMM hardware, which is hard to achieve at scale in a short term. Another solution to this end is soNUMA [50], which proposes to scale-out coherent NUMA memory interconnects at the datacenter level, therefore bringing cluster machines closer to each other. NeBuLa [59] similarly discusses the implementation of a hardware-offloaded RPC stack on top of soNUMA. NeBuLa introduces a novel mechanism for delivering RPC payloads directly into a processor's L1 cache, and also proposes an efficient in-LLC buffer management and load balancing method that reduces network queueing, and improves the tail latency of RPC requests. However, as with NetDIMM, both NeBuLa and soNUMA require fabrication of custom hardware that is physically integrated with the processor and memory subsystem, and reorganizing the entire datacenter network architecture, which is challenging to achieve at datacenter scale, especially given the frequent design and deployment changes of microservices. Another closely-related work, Optimus Prime [54], presents the design of an RPC data transformation accelerator which reduces the CPU pressure from expensive (de)serialization operations common in datacenter communication systems.

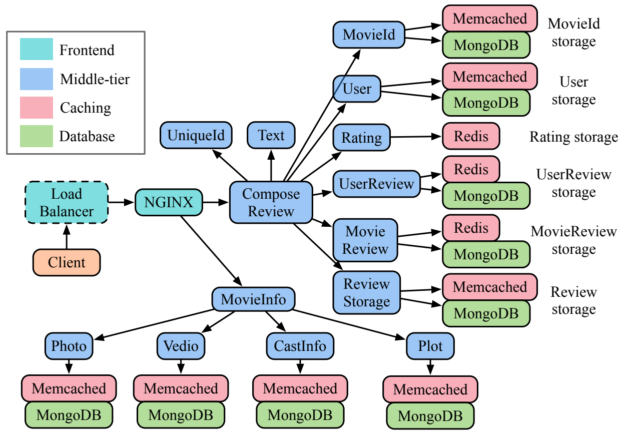


**Figure 2: Media Serving microservice architecture [33]** – client requests first reach a front-end load balancer, which evenly distributes them across the $N$ webserver instances. Then, depending on the type of user request, mid-tiers will be invoked to browse information about a movie, create a new movie review, or get recommendations on movies a user may enjoy. At the right-most of the figure, the requests reach the back-end databases, implemented both with in-memory caching tiers (memcached and Redis), and persistent databases (MongoDB).

At a high level, Dagger implements an RPC stack, fully offloaded to hardware. The FPGA-based design makes it amenable to the frequent updates present in microservices, and customizable to their diverse needs. In addition, Dagger leverages commercially-available memory interconnects as the interface between the processor and the *Ethernet* NIC, so, in contrast to previous solutions requiring custom hardware and/or specifically designed datacenter networks, our proposed system can be integrated into existing cloud infrastructures without the need for chip tapeouts. We also demonstrate that third party applications, such as memcached can be easily ported to Dagger, with minimal changes to the their codebase.

Even though Dagger's implementation is based on an FPGA, it is fundamentally different from other FPGA/SmartNIC-related proposals [21, 28, 29, 52] and industrial NICs. All existing networking devices are either "Smart" in the way they process packets but remain PCIe-attached, or are more closely-integrated to the main CPU but not programmable, only implementing simple packet delivery functionality. From this perspective, Dagger is the first attempt to leverage FPGAs which are *closely-coupled* with processors over memory interconnects as *programmable networking devices* to implement the networking stack all the way up to the application layer (RPCs). This new approach opens up many system challenges and opportunities to customize networking fabrics to the frequently-evolving interactive cloud services, in a high-performance and efficient manner. In addition, our proposal comes with out-of-the-box support for multi tenancy and co-location. We show how a single physical FPGA adapter can host multiple independent NICs serving different tenants running on the same host.

## 3 CHARACTERIZING NETWORKING IN MICROSERVICES

### 3.1 Networking Overheads in Microservices

We first study the network characteristics and requirements of interactive, cloud microservices. We use two end-to-end services from the DeathStarBench benchmark suite [33], shown in Figures 1 and 2; a Social Network, and a Media Serving application. We first profile the impact of the RPC stack on the per-tier and end-to-end latency for Social Network, for a representative subset of its microservices, shown in Figure 3. *s*1 is a `Media` tier processing embedded images and videos in user posts, *s*2 is the `User` tier responsible for managing a user's account and interfacing with the backend storage tiers, *s*3 is the `UniqueID` tier assigning a unique identifier to a new user post, *s*4 represents the `Text` service to add text to a new post, *s*5 is the `UserMention` tier to link to another user's account in a post, and finally *s*6 corresponds to the `UrlShorten` microservice, which shortens links embedded in user posts.

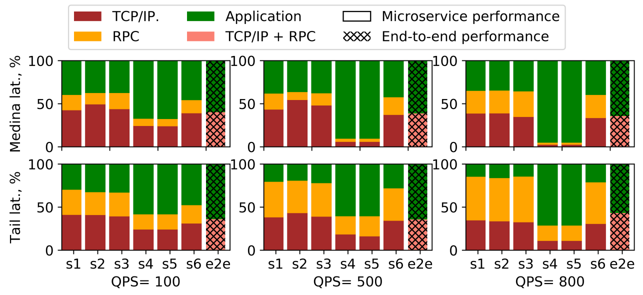


**Figure 3: Networking as a fraction of median (top) and 99th prc. tail (bottom) latency** – the input load increases progressively from the left-most to the right-most set of figures. The bars denote latency breakdown as recorded in the following individual microservices: s1: Media, s2: User, s3: UniqueID, s4: Text, s5: UserMention, s6: UrlShorten; e2e bar shows the breakdown for the end-to-end latency.

We break down communication overheads to RPC and TCP/IP processing, and show both median (top) and tail latency (bottom) for different load levels, from the left-most to the right-most figures. Across all tiers, communication accounts for a significant fraction of a microservice's latency, 40% on average, and up to 80% for the light in terms of computation `User` and `UniqueID` tiers. This also translates to a large fraction of end-to-end application latency going towards networking, despite many microservices overlapping with each other. The latency breakdown for the end-to-end application is shown in the right-most bar of each subfigure in Figure 3; communication accounts for at least third of the median and tail end-to-end latency. RPC processing is a large part of communication, and, for some services it is even larger than the TCP/IP stack when looking at tail latency, showing that accelerating the TCP/IP layer alone is not sufficient. A couple of microservices, such as `Text` and `UserMention` are more computationally intense, and therefore, their processing time is substantially longer than the communication latency. However, since microservices typically form deep call graphs, long delays in the upstream services can cause request queueing in downstream tiers, incurring cascading QoS violations [34].

The time devoted to networking increases considerably for higher loads (right plots in Figure 3), especially when looking at tail latency, due to excessive queueing across the networking stack. Not only does this degrade application performance, but the multiple layers of queueing in the network subsystem introduce performance unpredictability, making it hard for the application to meet its end-to-end QoS target. Note that since we cannot explicitly break down the time between queueing and processing at the application layer, for high loads, e.g., QPS=800, our profiler shows high percentages for RPC processing time; most of this time corresponds to queueing. We observe that the microservice's CPU utilization does not increase proportionally with load, in part because of back pressure from the downstream services. Such aggressive queueing in the RPC layer causes significant memory pressure and high memory interference with other tasks, making tail latency even more unpredictable and high. Overall, Figure 3 shows that networking, both the RPC layer and transport, in microservices is a major performance bottleneck, especially for tiers with a small amount of compute. The results are similar for the other end-to-end services as well

### 3.2 Network Characteristics of Microservices

We now analyse the network footprint of microservices. We profile the same Social Network and Media Serving applications [33], and show the histogram of their RPC sizes in Figure 4. The workload generator follows request distributions representative of real cloud services implemented with microservices [33]. Specifically, the request mix includes queries for users creating new posts (*Compose Post*), for reading their own timeline (*Read Home Timeline*, or another user's timeline *Read User Timeline*). Depending on the request type, a different subgraph of microservices is invoked [1].

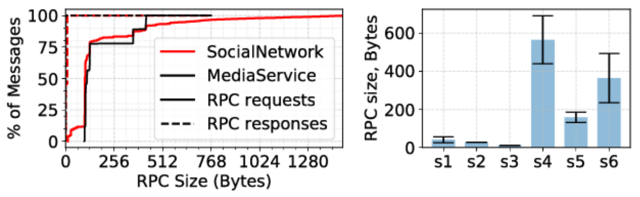


**Figure 4: Distribution of RPC request and response sizes in Social Network and Media Applications (left); breakdown of RPC sizes for individual microservices (right)** – s1 - s6 are the same as in Figure 3 for Social Network.

Figure 4 (left) shows the cumulative distribution function (CDF) of RPC sizes for each end-to-end service. 75% of all RPC requests are smaller than 512B. Responses are even more compact, with more than 90% of packets being smaller then 64B. Commodity networking systems experience poor performance when it comes to transferring small packets due to high per-packet overheads [38, 46]. This highlights the need for rethinking datacenter networking with the unique properties of microservices in mind.

Additionally, Figure 4 (right) shows that different microservices in the same application have very different RPC sizes. For example, the median RPC size in the *Text* service is 580B, while the *Media*, *User*, and *UniqueID* services never have RPCs larger than 64B. Such wide variation in RPC sizes across microservices shows that "one-size-fits-all" is a poor fit for microservice networking. Balancing between optimizing for large versus small packets is a long standing problem in networking. To achieve the best for this trade-off, Dagger employs reconfigurable hardware acceleration using FPGAs, to ensure that the networking stack can be tailored to the needs and characteristics of a given set of active microservices.

## 3.3 CPU Contention between Logic and Networking

The small size of microservices means that a server can concurrently host a large number of microservices, promoting resource efficiency. On the negative side, multi-tenancy also introduces resource interference, especially when CPU cores are used for both microservice logic and network processing. To quantify the resource contention between application logic and networking, we bind network interrupt service routines for each of the NIC's queues to a fixed set of N logical CPU cores ($N = 4$ in this experiment). We then run the Social Network service: (1) on the other $N$ CPU cores within the same NUMA node so that their execution does not interfere with networking, and (2) on the same $N$ cores as networking to observe the impact of interference. The resulting end-to-end request latency in each case is shown in Figure 5.

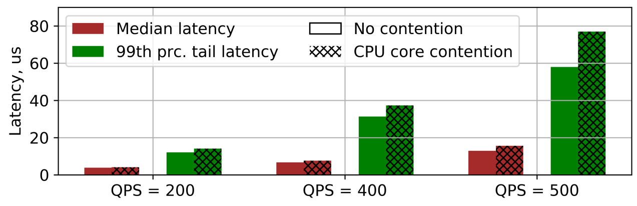


**Figure 5: Impact of the CPU resource interference between networking and application logic on end-to-end tail latency** – solid bars show latency when network processing and application logic run on different physical cores, while shaded bars show latency when network processing and application logic threads share the same CPU cores.

The experiment shows that when both application logic and network processing contend for the same CPU resources, end-to-end latency (both median and tail) suffers. As expected, interference becomes worse as the system load increases, especially when it comes to tail latency, which is more sensitive to performance unpredictability. Note that in addition to the network interrupt handling layer, RPC processing also interferes with application processing. Since the RPC stack is technically a part of the application logic in the service's default implementation, we are not able to isolate its impact; however, the high resource contention of Figure 5 already justifies the need for offloading the networking stack from the host processor. Dedicating, alternatively, cores specifically for network processing is not resource efficient, since network load fluctuates over time, dedicated networking cores are still prone to CPU-related overheads, and they can introduce interference in the last level cache (LLC) and main memory subsystems.

This analysis highlights three unique requirements for networking systems aimed at microservice deployments. First, RPC processing should be offloaded to dedicated hardware, to avoid CPU-related overheads and interference with the application logic. Second, systems should be optimized for small requests/responses, which dominate the network traffic of microservices. Finally, communication frameworks should be programmable to handle the diverse needs of microservices, and adjust to their frequent changes.

## 4 DAGGER DESIGN

### 4.1 High-Level Architecture and Design Principles

We design Dagger with the unique network properties and requirements of microservices in mind, discussed in Section 3. Although Dagger is optimized for microservices, it is still beneficial for traditional interactive cloud applications, as we will show in Section 5. Dagger's top-level architecture is shown in Figure 6. Dagger is based on three main design principles: i) *full hardware offload*, ii) *tight coupling*, and iii) *reconfigurability*.

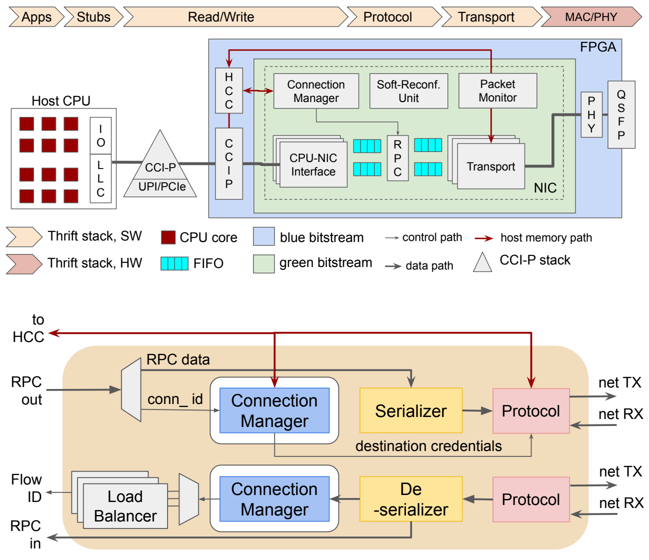


**Figure 6: Top-level architecture of Dagger (top) and zoom-in of the PRC unit (bottom)** – the bar on top shows approximate mapping of the Dagger stack onto the Thrift RPC [15] stack.

First, Dagger offloads the entire RPC stack in hardware, to eliminate all interference between application logic and network processing in the host CPU. The remaining software components of Dagger running on the host CPU are lightweight, and are only responsible for the connection set-up and for exposing the RPC API to applications. The software stack implements the API with zero-copying and directly places incoming RPC requests and responses to dedicated buffers (queues, rings) accessible by the hardware which is synthesized on an FPGA (shown by the NIC module in Figure 6). The rest of the processing is handled by the NIC.

The second design principle of Dagger is tightly coupling the hardware-accelerated network fabric with the host CPU. In contrast to existing high-performance and/or programmable NICs, which leverage PCIe-based interfaces, Dagger uses NUMA memory interconnects as the interface with the host processor to optimize the transfer of small fine-grained RPCs, by piggybacking on the hardware coherence protocol. The NUMA memory interconnect (Intel UPI in our current implementation) is encapsulated into the CCI-P protocol stack, as shown by the triangle in Figure 6. CCI-P [14] is a protocol stack between the host CPU and the FPGA designed by Intel, which in addition to UPI, also wraps-up two PCIe links.

Finally, the design of Dagger focuses on reconfigurability by leveraging an FPGA as the physical medium. Our current design is based on the Intel OPAE HDK that defines two regions of programmable logic: the blue and the green bitstreams. The former is fixed in the FGPA configuration and is not exposed to the FPGA users. This includes the implementation of system components, such as the CCI-P interface IP, Ethernet PHY, clock generation infrastructure, etc. The blue bitstream is managed by the cloud providers, and undergoes periodic updates. The green bitstream is used to implement the user logic and is fully programmable, with a pre-synthesized bitstreams. We implement the Dagger NIC in the green region, as shown in Figure 6. Our design is modular and configurable: different hardware parameters and components can be selected via SystemVerilog macros/parameters and synthesized. We call this *hard configuration*, and use it only for coarse-grained control decisions. For example, Dagger supports multiple different CPU-NIC interfaces; the choice of the specific scheme is performed via hard configuration, by selecting the corresponding IP blocks and configuring the design with them. Similarly, the choice of the transport layer, sizes of on-chip caches (e.g. connection caches), flow FIFOs, etc., are also enabled by hard configuration. Since hard configuration requires preparing dedicated bitstreams, and it incurs overheads to reprogram the FPGA, some fine-grained control decisions are still made via soft configuration. Soft configuration is based on soft register files accessible by the host CPU via PCIe MMIOs, and the corresponding control logic (Soft-Reconfiguration Unit in Figure 6). Dagger uses soft configuration to control the batch size of CCI-P data transfers, provision the transmit and receive rings, configure their number and sizes, configure the number of active RPC flows, choose a load balancing scheme, etc. Soft reconfiguration comes with certain logic overheads, however, it enables fast and fine-grained tuning of various parameters of the Dagger framework at runtime.

The remaining blocks shown in Figure 6 (top) are the Connection Manager used for setting up connections and storing all connection-related metadata, the Packet Monitor that collects various networking statistics, and auxiliary components, such as FIFOs, for proper synchronization of different blocks in the RPC pipeline. The Host Coherent Cache (HCC) is another important auxiliary unit in Dagger. HCC is a small (128 KB) direct-mapped cache implemented in the blue bitstream, which is fully coherent with the host's memory, via the CCI-P stack. HCC is used to hold cache connection states and the necessary structures for the transport layer on the NIC, while the actual data resides in the host memory. This way we avoid requiring dedicated DRAM hardware for the FPGA. This makes NIC cache misses cheaper compared to PCIe-based NICs, since the CCI-P stack provides hardware support for data consistency between the host DRAM and the HCC.

## 4.2 Dagger API and Threading Model

The API is designed following the standard client-server architecture of cloud applications, and is inspired by the Thrift RPC [15] framework as well as the Google Protocol Buffers interface, primarily for compatibility reasons, since many microservices rely on either of these APIs. Other RPC APIs, such as gRPC [13] or Finagle [5] can also be supported by the design. Similarly to commercial RPC stacks, Dagger comes with its own Interface Definition Language (IDL) and code generator. We adopt the Google Protobuf IDL for Dagger; an example of our interface definition scheme is shown in Listing 1.

```
Message GetRequest {   Message GetResponse {
  int32 timestamp;       int32 timestamp;
  char[32] key;          char[32] value;
}                      }

Service KeyValueStore {
  rpc get(GetRequest) returns(GetResponse);
  rpc set(SetRequest) returns(SetResponse);
}
```

**Listing 1: Dagger IDL on an example of a KVS service.**

The code generator parses target IDL files and produces client and server stubs which wrap up the low-level RPC structures being written/read to/from the hardware into the high-level service API function calls. The latter defines two main classes: the *RpcThreadedServer* and the *RpcClientPool* for each client-server pair. The *RpcClientPool* encapsulates a pool of RPC clients (*RpcClient*) that concurrently call remote procedures registered in the corresponding *RpcThreadedServer* as *RpcServerThread* objects wrapping server event loops and dispatch threads. Dagger supports both asynchronous (non-blocking) and synchronous (blocking) calls. In the former case, each *RpcClient* contains the associated *CompletionQueue* object which accumulates completed requests. The *CompletionQueue* might also invoke arbitrary continuation callback functions upon receiving RPC responses, if so desired.

The Dagger threading model is co-designed across hardware and software and is fully configurable, as is shown in Figure 7.

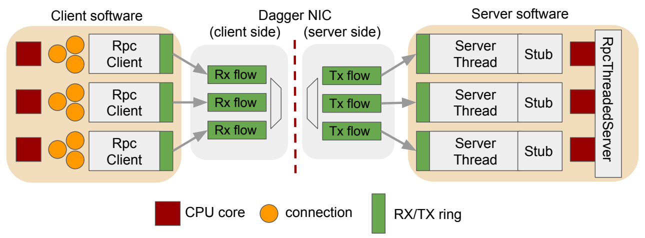


**Figure 7: Dagger threading model** – the green Rx/Tx flows correspond to hardware flows on the NIC (only single direction flows are shown for brevity; in the real system, each NIC runs both Rx and Tx flows).

Dagger provisions multiple flows (queues) on the NIC, such that each flow is 1-to-1 mapped to the corresponding RX/TX ring in software. The rings themselves are 1-to-1 mapped to *RpcClient*'s and *RpcServerThread*'s. The number of NIC flows, and therefore RX/TX rings, determines the degree of concurrency in the Dagger hardware, and is programmable via hard configuration. Note that the number of flows need not necessarily be equal to the number of CPU cores. However, in the basic scheme, shown in Figure 7, the number of NIC flows is decided based on the number of logical CPU cores, such that each core gets a dedicated parallel flow on the NIC. Similarly to FaRM, Dagger runs RPC handlers in dispatch threads to avoid inter-thread communication overheads. Also, with a small change in software, it can be configured to run RPC handlers in separate worker threads if required for long-running RPCs; this does not require any hardware changes.

Our threading model allows opening an arbitrary number of connections on each *RpcClient*. In this case, the connections on a certain *RpcClient* share the same RX/TX ring, so following the RDMA terminology, Dagger implements the Shared Receive Queue (SRQ) model [58]. Note that with the programmable threading model, Dagger can be configured to run in a single flow mode with a single RX/TX ring shared between multiple CPU cores. This enables models similar to [26] which target addressing load imbalance. At the other extreme, provisioning flows and rings on a per-connection basis is also possible, although such a scheme scales poorly and suffers from high load imbalance.

Dagger manages connections entirely in hardware which further reduces CPU load and improves the look-up of connection information for active flows. The NIC includes the Connection Manager (CM) module, as shown in Figure 6. The connection table interface maps connection IDs (c_id) onto tuples <src_flow, dest_addr, load_balancer>. The src_flow field specifies the ID of the flow receiving requests from the client. The NIC reads this information to ensure that the responses are steered to the same flows where requests came from. The dest_addr and load_balancer fields define the address of the destination host and preferred load balancing scheme for requests withing this connection.

The CM is designed as a simple direct-mapped cache with specific memory organization. In order to make the cache access concurrent and avoid stalls in the RPC flows, the cache breaks the above interface tuple into three tables indexed by the $\lceil log(N) \rceil$ LSBs (where N is the table size) of the connection ID providing 1W3R functionality. This is required because at the same time (cycle), three independent hardware agents might read from the cache: the RPC outgoing flow (to get the destination credentials), the incoming flow (to get the flow or load balancer), and the CM itself (to open and close connections). The size N of the cache is adjustable with hard configuration and can be chosen based on the expected number of connections the application might open. If some application requires many connections, N can be set to a high value giving more connection cache space to this application in favor of other NIC memory structures. Given the available size of FPGA on-chip memory (53Mb total minus 8.8Mb in the green region) and the size of the current connection tuple (8-12B)x3, the FPGA can be configured to cache at most 153K connections; sufficient for most application scenarios. In addition, the connection cache can be easily backed by DRAM (either externally attached to the FPGA or by the host DRAM) to allow more connections with certain performance penalty due to NIC cache misses. Although this functionality is not yet implemented in our current design, we plan to integrate it as part of future work (see the red lines in Figure 6).

## 4.3 NUMA Interconnects as NIC Interfaces

PCIe links have acted as the default NIC I/O interfaces for the past several decades. Despite the bus being a standard peripheral interconnect in any modern processor, a lot of prior work has shown that PCIe is not efficient as a NIC I/O interface [30, 49]. The inefficiency is mainly introduced in the transmission path, when the NIC is fetching network packets from the host memory. In the simplest case, commercial NICs use DMA transfers initiated by MMIO doorbell transactions to read packet descriptors and payloads from the software buffers; an approach known as the *doorbell method* [39].

However, the naïve doorbell scheme experiences inefficiencies when targeting small requests. The MMIO transactions are slow, mainly because they are implemented as non-cacheable writes, and expensive: every MMIO request should be explicitly issued by the processor. To reduce the overhead of MMIOs, modern high-performance NICs, such as Mellanox RDMA NICs, implement doorbell batching [39], an optimizations that allows grouping multiple requests into a single DMA transaction initiated with a single MMIO. While this solution noticeably increases the performance of doorbells, it still relies on MMIO messages and is only applicable when requests can be aggressively batched, which is not always possible for latency-sensitive flows. Another proposal [30] suggests eliminating DMAs, and transferring data only using MMIOs when requests fit in the MMIO's MTU, usually 1 cache line. This improves latency since data are transferred within a single transaction, however, performance is still limited by the low throughput of MMIOs, and during high load, this can overload the processor.

The fundamental limitation of PCIe protocols is that their design is primarily geared towards Producer-Consumer dataflow models [12]. The standard doorbell model works well under streaming flows and large data transfers. However, RPC requests do not always conform to such patterns. As we showed in Section 3, RPC sizes in microservices, and in other datacenter applications [46], are small, ranging from a few bytes and up to few kilo-bytes. In addition, the strict latency requirements of interactive services often disqualify batching, forcing NICs to handle fine-grained data chunks rather than streaming flows. This issue is further exacerbated when RPC frameworks do not just send requests, but also involve some amount of data processing. For example, Thrift RPC was designed to work with complex data objects that are not uniform in memory; for example, they might contain nested structures and references to other objects. In this case, RPCs must be (de)serialized [54], with existing PCIe models being very inefficient in fetching such non-uniformly placed objects. The standard doorbells used in all PCIe-attached NICs require expensive and CPU-inefficient data transformations before sending data to the NIC [54].

The main insight in leveraging memory interconnects as the NIC I/O is that they allow data transfers to be handled entirely in hardware. The memory consistency state machines (NUMA cache coherence protocols) implemented as a part of the processor's memory subsystem are designed to provide efficient and fast data flows between coherent agents; processors, or more generally, NUMA nodes. Making the NIC act as another NUMA node would allow it to closely integrate its I/O into the processor's memory subsystem, therefore providing a pure hardware CPU-NIC interface without the need for explicit notifications of data updates from the processor. This improves the CPU efficiency of sending small RPCs, since the only operation the processor needs to do is write the RPC requests/responses to the buffer it shares with the NIC, with the actual transfer handled entirely by the interconnect state machines. This increased CPU's efficiency significantly improves per-core RPC throughput (Section 5).

The exact scheme of data movements inside coherent busses depends on the specific NUMA interconnect model. Some specifications, such as the upcoming peripheral memory interconnect CXL [12], allow non-cacheable writes to the device memory, meaning that the CPU can directly write RPCs to the NIC, so in addition to improved CPU efficiency, the model also reduces latency, since only one bus transaction is required to send data to the device.

Note that we do not compare the *physical* performance of PCIe with respect to memory busses in this work. The peak bandwidth of both interconnects is implementation specific and depends on the number of lanes in the physical layer and the generation of the interconnect. Moreover, some memory interconnects are based on PCIe, so they use the same physical medium with the same bandwidth. All sources of performance gain shown in this work come from the difference in the *logical* upper-level communication models enabled by PCIe vs NUMA protocols. However, the theoretical bandwidth of the UPI bus which we use as the memory interconnect goes up to 19.2 GB/s; slightly higher than the 15.74 GB/s of the PCIe Gen3x16.

## 4.4 Implementation of the NIC Interface

The NIC I/O interface consists of the receiving and transmitting paths as seen from the NIC. The CPU-NIC interface diagram is shown in Figure 8.

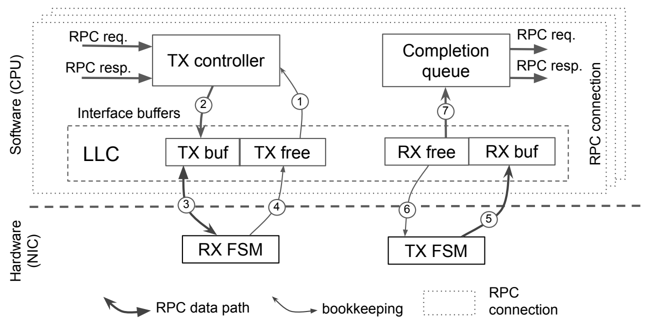


**Figure 8: CPU-NIC interface diagram** – RX path: TX controller writes new RPC requests/responses ② to a free entry in the TX buffer ①. The RX FSM on the FPGA fetches RPC objects from the TX buffer over CCI-P ③; it also does the bookkeeping ④ to release previously-fetched entries. TX path: the TX FSM puts newly-received RPC objects to the RX buffer, and asynchronously fetches the next free entries during bookkeeping ⑥. The RPC payload is finally copied to the completion queue ⑦.

Dagger provisions network buffers (RX/TX rings) on per-NIC flow basis. Each RX/TX pair reflects the communication channel between a single *RpcClient* and the corresponding *RpcThreadedServer* wrapping the dispatch thread. Such buffer provisioning enables lock-free access to the rings from the client and server threads [59]. As stated above, the same *RpcClient* can run multiple threads (each within a separate connection) therefore making them to share the corresponding RX/TX rings. In that case, explicit locking in the *RpcClient* RX/TX path is required to ensure consistent data transfer.

The RX/TX ring are used for incoming and outgoing RPCs, respectively. The stack is symmetric, the same architecture serves both requests and responses, i.e., the same NIC and the software stack can be used for both RPC clients and servers. Request types are distinguished by the *request type* field that is a part of every RPC packet. RX/TX rings are comprised of RX/TX buffers and free buffers. The former store RPC payloads for all requests until the NIC/completion queue acknowledges receiving the data by placing the ID of the corresponding RX/TX buffer entry into the free buffer. The size of the TX rings is determined by the rate of incoming RPCs and the time it takes for the NIC to fetch data and is configured during the NIC initialization. In our current prototype, the CCI-P-based memory interconnect, based on Intel UPI, delivers data from the software buffers to the NIC within 400 ns with another 400 ns required for sending back the bookkeeping information. The CCI-P bus can support up to 128 outstanding requests before reaching its bandwidth limit, so Dagger sends multiple asynchronous requests, while the bookkeeping information of in flight requests is pending. The size in the number of requests of the TX rings per flow is $\lceil Thr_{per\_flow} * 0.8/10^6 \rceil$, where $Thr_{per\_flow}$ is the desired throughput of the flow. For $Thr_{per\_flow}$ = 12.4 Mrps (the maximum throughput a single CPU core can sustain in our system), the size of the TX rings should be at least 10× the mean RPC size to avoid flow blocking. Given the typical sizes of RPCs in microservices, every flow requires 0.64-12.8 KB of TX buffers.

The RX rings accumulate a batch of requests before sending them to the completion queue. Therefore, the RX buffer size is equal to $B\times$ the mean RPC size, where $B$ is the width of CCI-P batching. In our experiments, the maximum sustained throughput is achieved with batching of $B = 4$. Below we detail the hardware design of the receiving and transmitting paths in Dagger.

*4.4.1 Receiving path (RX).* We implement three standard PCIe-based methods for fetching RPC data from the processor: doorbell, doorbell batching, and MMIO-based transfer, also known as WQE-by-MMIO in [39], alongside with our proposed method based on memory interconnects. The doorbell transfer is based on standard PCIe DMAs initiated via MMIO transactions. We use the DMA engine provided by the Intel OPAE HDK that operates over CCI-P. The doorbell batching is implemented by grouping CCI-P DMAs into single transactions and initiating the entire batch with a single doorbell. In the MMIO-based method, RPC requests and responses are transferred by writing them to a shared buffer allocated as an MMIO memory region on the FPGA. A typical MMIO write on most processors and NICs is a Write-Combine transaction. Write-Combine is used to avoid generating multiple PCIe transactions when writing data of a cache line size (64B), since processors normally commit word-aligned store instructions (8B). A Write-Combine buffer accumulates multiple stores and sends cache line-long chunks when the buffer is full or is being explicitly flushed. We modify this scheme with parallel store instructions based on the AVX-256 ISA extension and we do not use Write-Combining; this allows us to further reduce the MMIO latency. In this mode, Dagger writes every 64B of RPC requests using two _mm256_store_si256 stores.

The memory interconnect-based interface in Dagger is implemented over the Intel UPI coherent bus. Unfortunately, since CCI-P is the first commercially-available implementation of the UPI bus on an FPGA, the corresponding IP core, which is a part of the blue region of the FPGA bitstream and is therefore not accessible to users, only allows accessing CPU memory via memory polling. Given this limitation, Dagger starts by polling its local cache which is coherent with the processor's LLC and relies on invalidation messages to bring new data from software buffers. However, since the FPGA allocates data in its local cache in this case, it causes the CPU to lose ownership of the corresponding cache lines therefore hurting the data transfer's efficiency. For this reason, Dagger dynamically switches to direct polling of the processor's LLC when the load becomes high, as defined by a programmable threshold. Note that the current limitation of transferring data through polling is not fundamental to Dagger's design, but rather an implementation artifact of the currently-available realization of a memory interconnect on an FPGA. The next generation of FPGAs (Intel Agilex) already integrates a more advanced interconnect IP based on the CXL [12] specification. As mentioned above, the CXL protocol supports direct data writes from CPUs to the FPGA's memory without the need for memory polling.

*4.4.2 Transmitting path (TX).* The transmission path has some additional complexity compared to the RX path due to the need for load balancing and scheduling/distribution of flows over the active queues. Figure 9A shows the architecture of Dagger's TX path.

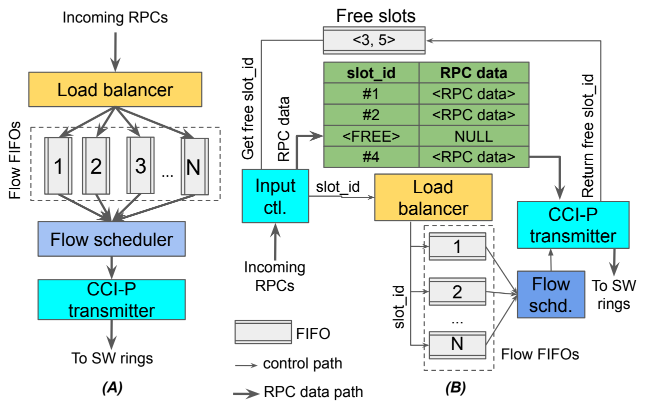


**Figure 9: (A) Architecture and (B) Implementation of TX path.**

Incoming RPCs should be distributed across the available receiving rings (RX buffers in Figure 8), and within a ring, the requests can be batched for more efficient data transfer, if a service's latency target permits it. Distribution of requests across rings is handled by the Load Balancer (as a part of the RPC unit, refer to Figure 6 (bottom)) that directs incoming RPCs to the corresponding Flow FIFOs. Each RX buffer gets a dedicated Flow FIFO on the NIC. The Load Balancer currently supports two request distribution schemes: dynamic uniform steering and static load balancing. In the first case, incoming RPCs are evenly distributed across the available flows. In the static balancing, the RPCs are distributed based on the information stored in the corresponding connection tuple on the server. In addition, we leave some room in the design for implementation of application-specific load balancers (e.g. the Object-Level core affinity mechanism in MICA [42]). The Flow Scheduler then picks a Flow FIFO that already contains enough requests to form a transmission batch and instructs the CCI-P transmitter to send the batch to the corresponding ring.

The architecture in Figure 9A is implemented in hardware, as shown in Figure 9B. Since Dagger's RPCs are at least 64 Bytes long, storing them in FIFOs for each flow and then multiplexing the flows is not practical, and adds complexity to the design. Instead, Dagger implements a request buffer, shown as the green table in Figure 9, which stores all incoming RPCs in a lookup table indexed by the slot_id. The Free Slot FIFO is designed to keep track of free entries in the request buffer. The Flow FIFOs in this case only contain references (slot_ids) to the actual RPC data in the table. When sending data, the CCI-P transmitter directly reads RPC payloads from the request table based on the references read from the corresponding Flow FIFO. The size of the request table is equal to $B * N_{flows}$ entries, where B is the CCI-P batching and $N_{flows}$ is the total number of NIC flows.

## 4.5 RPC Pipeline

The CPU-NIC interface is the first unit of the RPC pipeline; the other two being the RPC module and Transport layer (Figure 6). The architecture of the RPC module is zoomed-in in Figure 6 (bottom). It implements request serialization/de-serialization between ready-to-use RPC objects, as read from the processor and the network. If compression/encryption is required, the corresponding logic can optionally be integrated into the unit. In addition, the RPC module also contains a set of load balancers that decide which flow to steer incoming requests. The choice of the load balancer is controlled at server granularity, i.e. each server can specify the load balancer it requires when registering connections on it. The Protocol is the last module of the RPC unit. It is designed to implement RPC-optimized protocol layers such as congestion control, piggybacking acknowledgement, transactions built into the RC stack, etc., and is currently idle. Systems similar to TONIC [21] can be used to implement the Protocol unit.

The RPC unit is connected (over FIFOs, for synchronization) with the Transport layer which implements a version of the UDP/IP protocol and sends outgoing serialized RPC requests to the Ethernet network. Since data transformations and RPC transport protocols are not the focus of this work, we simplify these parts of the pipeline. Our current implementation only supports RPCs with continuous arguments that do not contain references to other objects and application-specific data-structures requiring custom serialization, and the Protocol unit is currently idle - it simply forwards all packets to the network. In the following-up work, we plan to extend Dagger with reliable transports and with RPC-specific congestion control mechanisms which has been shown to be more efficient in datacenter networks than TCP [38].

## 4.6 Dagger Implementation

We implement Dagger on an Intel Broadwell CPU/FPGA hybrid architecture. The host processor is a server-class Intel Xeon E5-2600v4

**Table 1: Implementation specifications of Dagger NIC.**

| Parameter | Value |
|---|---|
| CPU-NIC interface clock frequency, MHz | 200 - 300 |
| RPC unit clock frequency, MHz | 200 |
| Transport clock frequency, MHz | 200 |
| Max number of NIC flows [1] | 512 |
| FPGA resource usage, LUT (K) [2] | 87.1 (20%) |
| FPGA resource usage, BRAM blocks (M20K)[2] | 555 (20%) |
| FPGA resource usage, registers (K) [2] | 120.8 |

[1] Assuming 65K entries in the connection cache and ensuring the FPGA BRAM and logic utilization do not exceed 50%

[2] Including the blue region; UPI-based NIC I/O with 64 NIC flows and 65K entries in the connection cache

CPU integrated with an Arria 10 GX1150 FPGA. The hardware part (Dagger NIC) is written in SystemVerilog using the Intel OPAE HDK library. The software part is designed in C++11 and is compiled by GCC under under the O3 optimization level. The Dagger IDL code generator is written in Python 3.7. The software modules of our RPC stack run in user space, and the NIC buffers are allocated by the FPGA driver in the application virtual address space. The most important implementation parameters of the Dagger NIC design are summarized in Table 1.

### 4.7 Limitations

An important limitation of the current design is the lack of support for efficient RPC reassembling to enable transfer of requests larger than the cache line size. In contrast to PCIe DMA, memory interconnects implement relaxed memory consistency models, which is one of the key reasons behind their efficiency. Therefore, the MTU of a typical memory interconnect is only a single cache line [59]. A naïve solution to address the issue of sending larger RPCs is to reassemble requests in software. However, this will introduce CPU overheads and violate our first design principle. Another solution, as proposed in NeBuLa, leverages Content Addressable Memory (CAM) for on-chip reassembling in hardware. Unfortunately, CAMs are expensive in terms of area and energy, and it is challenging to implement them with low overheads on an FPGA. Efficient RPC reassembling in hardware is a challenging issue, and we plan to address it as part of future work. As of now, Dagger only features software-based RPC reassembling.

## 5 EVALUATION

### 5.1 Methodology

We evaluate Dagger along five dimensions. First, we compare Dagger with prior work on efficient RPC processing based on user-space networking and RDMA. Second, we evaluate different CPU-NIC interfaces and show the performance benefits of memory interconnects over PCIe. Third, we show how Dagger scales with the number of CPU threads. In addition, we demonstrate that Dagger can be easily integrated with existing datacenter applications, such as memcached and MICA, offering dramatic latency improvement under realistic workloads. Finally, we run a simple microservice application on top of Dagger showing that our RPC stack is indeed suitable for multi-tier systems. Table 2 shows the specification of the hardware platform used.

**Table 2: Hardware specifications of experimental platform.**

| **CPU: Intel Xeon E5-2600v4** | |
|---|---|
| Cores | 12 cores (OOO), 2 threads per core, 2.4 GHz, 14 nm |
| LLC | 30720 kB, 64 B |
| Additional features | AVX-2, DDIO, VT-x |
| OS | CentOS Linux, Kernel Linux 3.10.0 |
| **Interconnect: CCI-P: 2x PCIe and 1x UPI** | |
| PCIe | Gen3x8, 7.87 GB/s, 2 links, total bandwidth 15.74 GB/s |
| UPI | 9.6 GT/s (19.2 GB/s), 1 link, total bandwidth 19.2 GB/s |
| **FPGA: Arria 10 GX1150** | |
| Max frequency | 400 MHz |

Due to the limitation on the number of FPGA-enabled machines on the Intel vLab cluster, we instantiate two identical Dagger NICs on the same FPGA and connect them to each other via a loop-back network. We then give the NICs fair round-robin access to the CCI-P bus by multiplexing it. Note that since the main contribution of Dagger is in CPU-NIC interface and the RPC pipeline, the absence of physical networking does not affect our findings.

### 5.2 Performance Comparison across RPC Platforms

We compare the performance of Dagger's RPC acceleration fabric with four related proposals, based on DPDK user-space networking IX [23], raw user-space networking eRPC [38], RDMA FaSST [40], and the in-memory integrated NIC NetDIMM [18]. Table 3 shows the median round trip latency and the throughput achieved by each system. We also show the TOR (Top of Rack) delay assumed in each work, the size of the being transferred objects, and their type. Note that if the object type is "msg", this means that the system does not implement the RPC layers of the networking stack, and the reported results do not include the overhead of RPC processing.

**Table 3: Median round trip time (RTT) and throughput of single-core RPCs compared to related work [1].**

| | IX | FaSST | eRPC | Net-DIMM | Dagger |
|---|---|---|---|---|---|
| **Objects** | 64B msg | 48B RPC | 32B RPC | 64B msg | 64B RPC |
| **TOR delay** | N/A | 0.3 us | 0.3 us | 0.1 us | 0.3 us |
| **RTT, us** | 11.4 | 2.8 | 2.3 | 2.2 | 2.1 |
| **Thr., Mrps** | 1.5 | 4.8 [2] | 4.96 [2] | N/A | 12.4 |

[1] Performance numbers are provided from corresponding papers

[2] Recorded in symmetric experimental settings

As seen from Table 3, Dagger shows 1.3 – 2.5× (depending on experimental settings) higher per-core RPC throughput than the RDMA-based solution, FaSST, and the DPDK-based eRPC. The gain partially comes from offloading the entire RPC stack on hardware and leaving only a single memory write in the critical RPC path on the processor. In addition, approximately 14% of performance improvement is enabled by replacing the doorbell batching model with our memory interconnect-based interface. Note that neither PCIe's nor UPI's physical bandwidth is saturated in this experiment, so these 14% come from the better messaging model enabled by memory interconnects. Table 3 also shows that Dagger achieves the lowest median round trip time of 2.1*us*, while significantly improving throughput compared to both user-space and kernel-level networking. This is better than FaSST, and even the integrated solution NetDIMM, and is comparable with eRPC.

## 5.3 Comparison of CPU-NIC Interfaces

Figure 10 shows the comparison of Dagger's end-to-end single-core latency and throughput for different CPU-NIC interfaces. Unless otherwise specified, the PCIe CPU-NIC interface is based on a single PCIe Gen3x8 link. The maximum theoretical (physically bounded) throughput is 122 Mrps for 64 Byte RPCs on all cores.

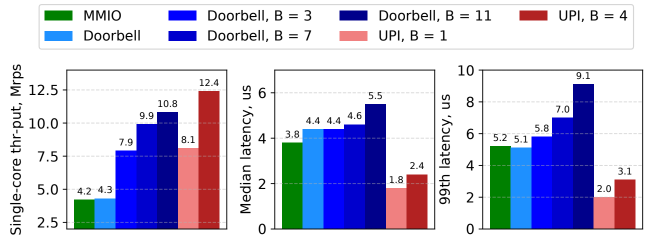


**Figure 10: Dagger's single-core throughput and latency for different CPU-NIC interfaces (RX path) when transferring 64 Bytes RPCs** – B denotes request batching.

As seen from Figure 10, the lowest median and tail latency over a PCIe bus is achieved when the RX path uses MMIO writes, which is reasonable, since in this case all RPCs are being written within a single PCIe transaction. However, the method fails to deliver high throughput, with the best reported result being 4.2 Mrps. A similar throughput of 4.3 Mrps but with higher median latency is reported when Dagger is using non-batched doorbells showing that their performance is limited by the rate of initiating MMIOs. The only way to increase the efficiency of doorbells is to use batching which enables reaching a throughput of 10.8 Mrps for batch of $B = 11$. The memory interconnect-based transfer (UPI in Figure 10) achieves single-core throughput of 12.4 Mrps with $B = 4$, and demonstrates noticeably lower median and tail latency. Note that the latency improvement does not exclusively come from the reduced number of bus transactions for the transfer using the memory interconnect. We conduct another experiment in which we access an address in the shared memory over PCIe (using DMA) and over UPI. The PCIe DMA gives us 450*us* of median one-way latency while the UPI read achieves 400*us*. This shows that UPI is physically slightly faster than PCIe. Finally, we measured the maximum single-core throughput of Dagger at 16.5 Mrps, with best-effort request processing by allowing arbitrary packet drops by the server.

## 5.4 Latency vs Throughput

Figure 11 (left) shows Dagger's latency under different loads with the memory interconnect-based NIC.

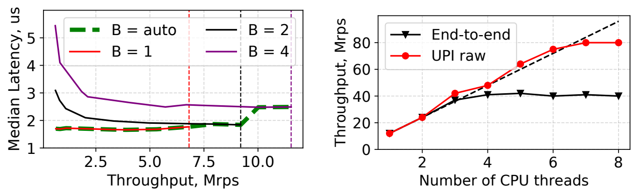


**Figure 11: Latency-Throughput curves for single-core asynchronous round-trip 64B RPCs (left) and multi-core scalability of sending 64B requests (right)** – B denotes batching, dotted lines show the saturation point. The black line shows end-to-end RPC throughput, the dashed line denotes estimated linear scalability, and the red line shows the results of the UPI bus's scalability with raw idle requests.

With CCI-P batching $B = 1$, Dagger achieves the lowest median latency (round trip time) of 1.8*us*, which remains stable across the entire load range, up to the throughput saturation point of 7.2 Mrps. When increasing batching to $B = 4$, Dagger's throughput increases to 12.4 Mrps with a latency of 2.8*us*. Note that when the load is low, the request latency is relatively high since the RPC pipeline needs to wait until the batch is full. Dagger leverages soft configuration to adjust the batch size dynamically when the load becomes high so that the throughput advantages of batching do not come at a latency cost (as shown by the green dashed line in Figure 11 (left)).

## 5.5 Thread Scalability

Figure 11 (right) plots the scalability of Dagger with the number of CPU threads (logical cores). The system throughput scales linearly up to 4 threads (2 physical cores) and remains flat at 42 Mrps. Note that since we run both the RPC client and server on the same CPU, this effectively translates to 84 Mrps as seen by the processor. This result is ≈ 23% higher than the performance reported in the FaSST paper on the CIB cluster, and is 3.5× better than IX under the same number of cores. The saturation results signal that the current bottleneck is not the processor. The NIC itself, which is capable of processing up to 200 Mrps, is also far from saturated. To better understand the scalability of Dagger, we run another experiment in which we send idle memory read requests over the UPI interconnect, the results of which are shown in Figure 11 (right) in red. The throughput of idle memory reads also scales linearly up to 80 Mrps with 7 threads, and stays flat when one more thread is added. Note that the UPI bus has the total physical bandwidth of 19.2 GB/s (Table 2) which is significantly more than 80Mrps for 64 Byte RPCs. Based on this experiment, we conclude that the current bottleneck is the implementation of the UPI end-point on the FPGA in the blue region. Since this region is encrypted, we are not able to optimize it in the current prototype, however, the upcoming generation of Intel Agilex FPGAs will have a dedicated hard IP core for memory interconnects, which should address this scalability issue.

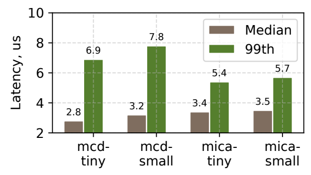
## 5.6 End-to-End Evaluation on KVS systems

We evaluate the end-to-end performance of Dagger on two real KVS systems: memcached [6] and MICA [42]. Memcached is a popular in-memory key-value store, widely used in microservices [33, 54, 55].

In this experiment, we run the original version of memcached over Dagger with the UPI-based I/O instead of the native transport protocol based on TCP/IP for SET and GET commands. We modify only ≈50 LOC of the Memcached source code in order to integrate it with Dagger. We also keep the original memcached protocol to verify the integrity and correctness of the data.

While memcached is a suitable application because of its wide adoption in both industry and academia, it is relatively slow (≈12× slower than Dagger). Therefore, the performance of memcached over Dagger is bottlenecked by memcached itself, which does not allow Dagger to achieve its full potential. In order to make the evaluation more comprehensive, we also port MICA over Dagger, another well-known KVS system in the academic community. MICA is designed specifically to provide high throughput for small requests, and, in contrast to memcached, it can be, under certain workloads, network-bounded. We run MICA over Dagger with no changes to the original codebase; we simply implement a MICA server application which integrates it with Dagger with ≈200 LOC.

To evaluate the KVS systems, we generate two types of datasets similar to the ones used to evaluate MICA [42]: tiny (8B keys and 8B values) and small (16B keys and 32B values). We populate both memcached and MICA KVS with 10M and 200M unique key-value pairs respectively, and access them over the Dagger fabric, following a Zipfian distribution [22] with skewness of 0.99. For MICA, we use their original benchmark and workload generator [9]. We load both systems with two types of workloads: write-intense (set/get = 50%/50%) and read-intense (set/get = 5%/95%). We adjust the workload generator such that the number of packet drops on the server is always < 1%. The results of running memcached and MICA with Dagger on a single core are shown in Figure 12.



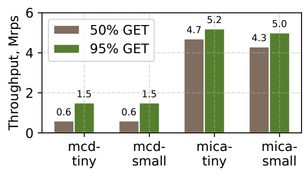


**Figure 12: Performance of memcached and MICA running over Dagger: request latency (left) and throughput (right) –** latency results recorded under the write-intensive workload and the peak single-core throughput.

Figure 12 (left) shows the latency of the KVS systems running over Dagger. Latency is defined as the round-trip time between the moment when a request is issued by a core and when the result is received. We measure latency for the write-intensive (set/get = 50%/50%) workload in a similar way to [42] to ensure a fair comparison. Memcached achieves the lowest median latency of $2.8us$, and a $99^{th}$ percentile latency of $6.9us$, when running under a $0.6Mrps$ load on a single core for tiny requests. For small requests, the results are slightly higher. The MICA KVS shows a better tail latency of $5.4us$ and $5.7us$ for tiny and small requests, respectively. It is $4.4 - 5.2\times$ lower than the latency numbers reported in the original MICA work when running it over the lightweight DPDK-based networking stack. This result shows that Dagger can offer dramatic latency reduction in high performance KVS systems compared to optimized user-space networking.

The KVS throughput results are shown in Figure 12 (right). With the workload that we use, the systems are still bottlenecked by the key-value store. Dagger can reach up to $12.4Mrps$ of single-core throughput, where memcached and MICA had a limit of $0.6 - 1.6$ and $4.8 - 7.8$ Mrps, respectively. For this reason, the single-core throughput improvement of Dagger remains hidden. At the same time, these results show that integrating our RPC stack does not add any additional throughput overhead compared to software-based networking systems. To better load MICA, we also test it under a distribution with skewness of 0.9999, which yields even higher data locality, and therefore better cache utilization. With such a workload, Dagger achieves a throughput of 10.2 Mrps and 9.8 Mrps for read- and write-intensive workloads with the same latency numbers, as in Figure 12, therefore bringing the performance of MICA closer to the peak performance of Dagger.

We do not show results of multi-core scalability for MICA, since the extensive amount of LLC contention introduces considerable instability in the results. The contention is caused by running both client and server on the same CPU, without the possibility to partition the cache, therefore allowing them to share the same portion of LLC. The client runs the workload generator which reads 1.49GB of data at a very high rate from its internal buffer making the LLC traffic very high. As part of future work, we plan to deploy Dagger to a cluster environment with physically distributed FPGAs to avoid client-server colocation, and measure more representative multi-core throughput results.

## 5.7 End-to-End Evaluation on Microservices

Finally, we evaluate Dagger on an end-to-end application built with microservices. We design a simple multi-tier service with 8 microservices which implements a Flight Registration service, shown in Figure 13.

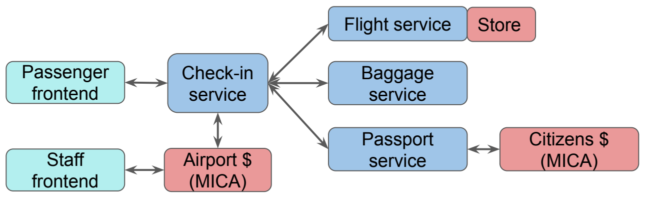


**Figure 13: Flight Registration microservice architecture** – the passenger front-end generates uniformly random passenger registration requests to the Check-in service. The Check-in service then consults the Flight service for flight information data, the Baggage service for the status of the passenger's baggage, and the Passport service to check the passenger's identity. The Passport service issues nested requests to the Citizens database (based on MICA [42]). Upon receiving all responses, the Check-in service registers the passenger in the Airport database (also based on MICA cache). The latter is additionally accessible by the Staff front-end, which is used to asynchronously check all records in the system.

We design the Flight Registration service such that it exhibits different types of dependencies across tiers (one-to-one, one-to-many, many-to-one), and includes both chain and fanout dependencies. All services communicate with each other over RPC calls and run different threading models to show the flexibility of Dagger. In particular, the *Passenger* and *Staff Frontend* services run non-blocking RPCs to avoid throughput bottlenecks due to high request propagation times. Similarly, the *Check-in* service issues non-blocking requests to the *Flight*, *Baggage*, and *Passport* services, but it later blocks until it receives all the responses before proceeding with blocking calls to the *Airport* service. The *Passport* service also runs blocking RPCs to the Citizens database. In our first experiment, each RPC server processes requests in the dispatch thread to benefit from its low-latency, zero-copy operation.

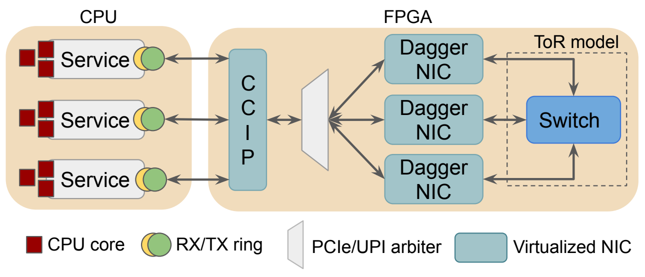


**Figure 14: Virtualization of the Dagger NIC to serve multiple tiers running on the same physical machine** – the PCIe/UPI arbiter provides fair round-robin sharing of the CCI-P bus between tenants; the Switch performs simple L2 packet switching based on the pre-defined static switching table.

Similarly to our previous experiments, and due to the limitations of the cluster, we run all services on the same machine and over the same physical FPGA. Such a setup requires virtualization pf the NIC to ensure that each tier gets an independent Dagger NIC therefore reflecting the real distributed setup when each tier runs on a separate physical or virtual machine. By placing all tiers on the same machine we additionally show how the Dagger NIC can be virtualized. Our NIC is virtualizable by putting multiple instances of it on the same FPGA and giving them fair round-robin sharing of the system bus and memory. Each instance of the NIC is serving a dedicated microservice tier, i.e., in this experiment, we instantiate 8 copies of the Dagger NIC, as shown in Figure 14. The NICs are connected with each other over our simple model of a ToR networking switch with a static switching table. In Section 6, we give more details on virtualization and show how different microservice tiers can benefit from it.

As previously mentioned, we first run the Flight Registration service with the *Simple* threading model in which each service handles RPC requests directly in dispatch threads along with the networking I/O. The results of this experiment are summarized in the first row in Table 4.

**Table 4: Summary of performance results for the Flight Registration service.**

| Threading model | Highest load, Krps [1] | Lowest latency, us | | |
|---|---|---|---|---|
| | | Median | 90th | 99th |
| Simple | 2.7 | 13.3 | 20.2 | 23.8 |
| Optimized | 48 | 23.4 | 27.3 | 33.6 |

[1] Recorded when the total number of request drops across all tiers does not exceed 1%.

As seen from Table 4, the maximum throughput with the *Simple* threading model is limited to 2.7$Krps$, however, the system shows low $\mu s$-scale end-to-end latency. In order to profile the application, we design a lightweight request tracing system and integrate it with Dagger. Our analysis reveals that the system is bottlenecked by the resource-demanding and long-running *Flight* service. Handling such RPCs in dispatch threads limits the overall throughput since they block the NIC's RX rings from reading new requests. One well-known way [51] to address this issue is to use different dispatch and worker threads for networking IO and RPC handling. Similar intuition applies to the *Check-in* and *Passport* services. Those are not resource-intensive, but they issue multiple nested blocking RPC calls, and therefore run for a relatively long time. In our next experiment, we configure the *Flight*, *Check-in*, and *Passport* service's RPC servers to run request processing in worker threads (*Optimized* threading model). The results in the second row in Table 4 show that such a change in the threading model dramatically increases the overall application throughput by up to 17×. Note that the latency became larger in this case due to the overhead of inter-thread communication and additional request queueing between the dispatch and worker threads. Figure 15 shows a more comprehensive view onto the system behaviour when running with the *Optimized* threading model. The 8-tier Flight Registration service running over Dagger achieves a median and tail latency of 23 and 33$us$ respectively before the throughput saturation point of 25$Krps$. When throughput crosses the saturation point, the tail latency soars sharply, while the median latency stays at the level of 23 – 26$us$. We do not increase the load further due to the quickly growing number of request drops after that point.

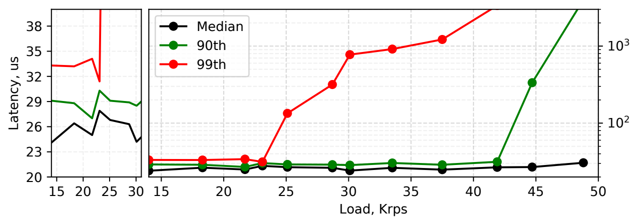


**Figure 15: Latency/load curves for the Flight Registration service with the *Optimized* threading model** – the figure on the left shows a zoomed-in view of the latency over the load up to 30 Kqps in linear scale.

In addition to configuring the software part of Dagger to run different threading models, we also configure the NICs differently. All services in our Flight Registration benchmark besides the *Airport* and *Citizens* services are stateless and they do not cache any data. For this reason, the round-robin dynamic RPC load balancer works best for them and we configure the corresponding NICs to use it. In contrast, the two mentioned services are stateful, and they run the MICA KVS in the backend. However, MICA does not work correctly with round-robin/random load balancers due to the way it partitions the object heap across CPU cores/NIC flows. MICA requires that all requests with the same keys always go to the same partition, and in the original work, it uses Flow Director to steer requests to cores/partitions. In this experiment, we implement our own application-specific Object-Level [42] load balancer for MICA tiers by applying the hash function to each request's key on the FPGA before steering them to the flow FIFOs, and instantiate it inside the Dagger NICs serving the corresponding tiers. This shows how NICs running hardware-offloaded RPCs on FPGAs can be flexibly programmable to satisfy the needs of different applications. More hardware parameters of the NICs can be further fine-tuned for each individual microservice as we briefly discuss in Section 6.

## 6 DISCUSSION

As shown in Dagger's evaluation, relying on memory interconnects for high performance datacenter networking is beneficial, however one might argue that integrating FPGAs/NICs over conventional PCIe busses imposes fewer constraints over the type of CPU a system can use. Additionally, while PCIe is more widely adopted as a peripheral interconnect in today's processors, this trend is increasingly changing. First, the UPI/QPI interconnect is natively supported by all modern datacenter processors (e.g., Xeon family), and any FPGA which implements the UPI/QPI specification can be integrated in it. As of today, we are aware of two such FPGA families: Intel Broadwell Arria 10 (used in this work) and the new Stratix 10 DX device. A similar technology is also being developed by IBM. Their OpenCAPI [10] cache-coherent CPU-FPGA interface is already used in recent research work on disaggregated memory [53]. Second, there is an ongoing collaborative effort from multiple hardware vendors to derive a specification for a new peripheral interface with cache coherency support (CXL) for future devices. Similar efforts in academia have yielded systems like ETH's Enzian [8], which closely couples an FPGA with an ARM-based datacenter CPU over the Cavium coherent interconnect CCPI [7]. We believe, Enzian can also be a good physical medium for Dagger.

Virtualization of network interfaces is another topic around Dagger's design. NIC virtualization enables efficient sharing of a single physical interface between multiple tenants, such as different guest operating systems. Given that Dagger is based on an FPGA and it can be tuned for different applications based on their network characteristics and requirements, it provides an excellent framework to enable virtualization. As seen from Table 1, the Dagger NIC occupies less than 20% of the available FPGA space when synthesized with a relatively large number of flows and connection cache space. This demonstrates that the same FPGA device can accommodate multiple instances of the NIC at the same time as we show in Section 5.7. Each instance can be used as a "virtual but physical" NIC for the corresponding tenant, and it can be configured based on the network provisioning requirements of each tenant.

Additionally, we note that the BRAM memory of FPGAs is one of the key resources enabling reconfigurability and efficiency in Dagger. By leveraging the FPGA to manage on-chip memory one can flexibly split the available memory capacity (53Mb for the FPGA used in Dagger) at very fine granularity, therefore improving the efficiency of NIC caches which is crucial, since NIC cache misses are one of the main performance bottlenecks in commercial NICs [39]. This is especially important in the aforementioned virtualized environment. For example, with FPGAs, it is possible to allocate more connection cache memory for NIC instances serving tenants with a large number of connections, or more packet buffer space for tenants experiencing large network footprints. Such on-chip NIC cache management can be easily done at the NIC instance granularity.

Finally, Dagger-like designs enable efficiently co-designing RPC stacks with transactions in hardware. For instance, in our example of the Flight Registration application in Section 5.7, the *Airport* service concurrently processes requests from both the *Check-in* service and *Staff Frontend*. As of now, we implement a lock-based concurrency control mechanism in software which comes with certain overheads in the OS. Alternatively, given the fully programmable nature of FPGAs, one can run synchronization protocols at the RPC level, on the Dagger NIC, such that all requests being received by the service are already serialized. Applications with more complicated transactional semantics (e.g., Paxos, 2PC, etc.) can specifically benefit from this support.

## 7 CONCLUSION

Dagger is an efficient and reconfigurable hardware acceleration platform for RPCs, specifically targeting interactive cloud microservices. In addition to showing the benefits of hardware offload for RPCs to reconfigurable FPGAs, we also demonstrate that using memory interconnects instead of PCIe as the NIC I/O interface offers significant benefits. Most importantly, our work shows that such close coupling of programmable networking devices with processors is already feasible today. Dagger achieves 1.3 – 3.8× better per-core RPC throughput compared to previous DPDK- and RDMA-based solutions, it provides $\mu$-scale latency, and it can be easily ported to third-party applications, such as memcached and MICA, significantly improving their median and tail latencies. We also show the reconfigurability feature of our proposal by running an example of a multi-tier application and tuning both the software and hardware parts of the stack for each individual microservice to get high end-to-end performance.

## ACKNOWLEDGMENTS

We sincerely thank Lizy John for her valuable feedback while shepherding our paper. We also thank Yu Gan, Yanqi Zhang, Shuang Chen, Neeraj Kulkarni, Mingyu Liang, Zhuangzhuang Zhou, Mark Sutherland, and the anonymous reviewers for their feedback on earlier versions of this manuscript. This work was in part supported by CRISP, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA, an NSF CAREER Award CCF-1846046, NSF grant NeTS CSR-1704742, NSF/Intel CAPA grant CCF-1723715, a Sloan Research Fellowship, a Microsoft Research Fellowship, an Intel Faculty Rising Star Award, and a John and Norma Balen Sesquisentennial Faculty Fellowship.

## REFERENCES

[1] [n.d.]. DeathStarBench Github Repository. https://github.com/delimitrou/DeathStarBench.
[2] [n.d.]. Decomposing Twitter: Adventures in Service-Oriented Architecture. https://www.slideshare.net/InfoQ/decomposing-twitter-adventures-in-serviceoriented-architecture.
[3] 2016. The Evolution of Microservices. https://www.slideshare.net/adriancockcroft/evolution-of-microservices-craft-conference.
[4] accessed August, 2020. Cloudius Systems. Seastar. (accessed August, 2020). http://www.seastarproject.org/.
[5] accessed August, 2020. Finagle RPC. (accessed August, 2020). https://twitter.github.io/finagle/.

[6] accessed August, 2020. Memcached source. (accessed August, 2020). https://github.com/memcached/memcached.
[7] accessed December, 2020. Cavium CCPI interface. (accessed December, 2020). https://en.wikichip.org/wiki/cavium/ccpi.
[8] accessed December, 2020. ETH Enzian research computer. (accessed December, 2020). http://www.enzian.systems/.
[9] accessed December, 2020. MICA source code. (accessed December, 2020). https://github.com/efficient/mica.
[10] accessed December, 2020. The OpenCAPI consortium. (accessed December, 2020). https://opencapi.org/.
[11] accessed July, 2020. DPDK. (accessed July, 2020). https://www.dpdk.org/.
[12] accessed May, 2020. Compute Express Link (CXL) specification. (accessed May, 2020). https://www.computeexpresslink.org/.
[13] accessed May, 2020. gRPC. (accessed May, 2020). https://grpc.io/.
[14] accessed May, 2020. Intel Acceleration Stack for Intel Xeon CPU with FPGAs Core Cache Interface (CCI-P) Reference Manual. (accessed May, 2020). https://www.intel.com/content/www/us/en/programmable/documentation/buf1506187769663.html.
[15] accessed May, 2020. Thrift RPC. (accessed May, 2020). https://thrift.apache.org/.
[16] Adrian Cockroft [n.d.]. Microservices Workshop: Why, what, and how to get there. http://www.slideshare.net/adriancockcroft/microservices-workshop-craft-conference.
[17] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. 2003. Performance Debugging for Distributed Systems of Black Boxes. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, USA) *(SOSP '03)*. Association for Computing Machinery, New York, NY, USA, 74–89. https://doi.org/10.1145/945445.945454
[18] Mohammad Alian and Nam Sung Kim. 2019. NetDIMM: Low-Latency Near-Memory Network Interface Architecture. *Int'l Symp. on Microarchitecture (MICRO)* (2019).
[19] Mohammad Alizadeh, Adel Javanmard, and Balaji Prabhakar. 2011. Analysis of DCTCP: Stability, Convergence, and Fairness. *Int'l Conf. on Measurement and Modeling of Computer Systems (SIGMETRICS)* (2011).
[20] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. 2013. PFabric: Minimal near-Optimal Datacenter Transport. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM* (Hong Kong, China) *(SIGCOMM '13)*. Association for Computing Machinery, New York, NY, USA, 435–446. https://doi.org/10.1145/2486001.2486031
[21] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. 2020. Enabling Programmable Transport Protocols in High-Speed NICs. *USENIX Symp. on Networked Systems Design and Implementation (NSDI)* (2020).
[22] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload Analysis of a Large-Scale Key-Value Store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems* (London, England, UK) *(SIGMETRICS '12)*. Association for Computing Machinery, New York, NY, USA, 53–64. https://doi.org/10.1145/2254756.2254766
[23] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. [n.d.]. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. *USENIX Symp. on Operating Systems Design and Implementation (OSDI)* ([n. d.]).
[24] Adrian Caulfield, Paolo Costa, and Monia Ghobadi. 2018. Beyond SmartNICs: Towards a Fully Programmable Cloud: Invited Paper. (2018).
[25] P. Chen, Y. Qi, P. Zheng, and D. Hou. 2014. CauseInfer: Automatic and distributed performance diagnosis with hierarchical causality graph in large distributed systems. In *IEEE INFOCOM 2014 - IEEE Conference on Computer Communications*. 1887–1895.
[26] Alexandros Daglis, Mark Sutherland, and Babak Falsafi. 2019. RPCValet: NI-Driven Tail-Aware Balancing of Ms-Scale RPCs. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2019).
[27] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast Remote Memory. *USENIX Symp. on Networked Systems Design and Implementation (NSDI)* (2014).
[28] Haggai Eran, Lior Zeno, Maroun Tork, Gabi Malka, and Mark Silberstein. 2019. NICA: An Infrastructure for Inline Acceleration of Network Applications. *USENIX Annual Technical Conf. (ATC)* (July 2019).
[29] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation* (Renton, WA, USA) *(NSDI'18)*. USENIX Association, USA, 51–64.
[30] Mario Flajslik and Mendel Rosenblum. 2013. Network Interface Design for Low Latency Request-Response Protocols. *USENIX Annual Technical Conf. (ATC)* (2013).
[31] Yu Gan and Christina Delimitrou. 2018. The Architectural Implications of Cloud Microservices. In *Computer Architecture Letters (CAL), vol.17, iss. 2.*
[32] Yu Gan, Mingyu Liang, Sundar Dev, David Lo, and Christina Delimitrou. 2021. Sage: Leveraging ML To Diagnose Unpredictable Performance in Cloud Microservices. In *Proceedings of the Twenty Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).*
[33] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud and Edge Systems. *International Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2019).
[34] Yu Gan, Yanqi Zhang, Kelvin Hu, Yuan He, Meghna Pancholi, Dailun Cheng, and Christina Delimitrou. 2019. Seer: Leveraging Big Data to Navigate the Complexity of Performance Debugging in Cloud Microservices. In *Proceedings of the Twenty Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Providence, RI).
[35] Stephen Ibanez, Muhammad Shahbaz, and Nick McKeown. 2019. The Case for a Network Fast Path to the CPU. *ACM Workshop on Hot Topics in Networks* (2019).
[36] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. 2014. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. *USENIX Symp. on Networked Systems Design and Implementation (NSDI)* (2014).
[37] Vimalkumar Jeyakumar, Omid Madani, Ali Parandeh, Ashutosh Kulshreshtha, Weifei Zeng, and Navindra Yadav. 2019. ExplainIt! – A Declarative Root-Cause Analysis Engine for Time Series Data. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 333–348. https://doi.org/10.1145/3299869.3314048
[38] Anuj Kalia, Michael Kaminsky, and David Andersen. 2019. Datacenter RPCs can be General and Fast. *USENIX Symp. on Networked Systems Design and Implementation (NSDI)* (2019).
[39] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. Design Guidelines for High Performance RDMA Systems. *USENIX Annual Technical Conf. (ATC)* (2016).
[40] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. *USENIX Symp. on Operating Systems Design and Implementation (OSDI)* (2016).
[41] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr. Sharma, Arvind Krishnamurthy, and Thomas Anderson. 2019. TAS: TCP Acceleration as an OS Service. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 24, 16 pages. https://doi.org/10.1145/3302424.3303985
[42] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. 2014. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. *Symposium on Networked Systems Design and Implementation (NSDI)* (2014).
[43] JinJin Lin, Pengfei Chen, and Zibin Zheng. 2018. Microscope: Pinpoint performance issues with causal graphs in micro-service environments. In *International Conference on Service-Oriented Computing*. Springer, 3–20.
[44] Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. 2019. E3: Energy-Efficient Microservices on SmartNIC-Accelerated Servers. *USENIX Annual Technical Conf. (ATC)* (2019).
[45] L. Ming, Qiulei Fu, Y. Wan, and T. Zhu. 2012. User-space RPC over RDMA on InfiniBand.
[46] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. 2018. Homa: A Receiver-Driven Low-Latency Transport Protocol Using Network Priorities. *ACM Special Interest Group on Data Communication (SIGCOMM)* (2018).
[47] YoungGyoun Moon, SeungEon Lee, Muhammad Asim Jamshed, and KyoungSoo Park. 2020. AccelTCP: Accelerating Network Applications with Stateful TCP Offloading. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 77–92. https://www.usenix.org/conference/nsdi20/presentation/moon
[48] Akshay Narayan, Frank Cangialosi, Deepti Raghavan, Prateesh Goyal, Srinivas Narayana, Radhika Mittal, Mohammad Alizadeh, and Hari Balakrishnan. 2018. Restructuring Endpoint Congestion Control. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (Budapest, Hungary) *(SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA, 30–43. https://doi.org/10.1145/3230543.3230553
[49] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. 2018. Understanding PCIe Performance for End Host Networking. *ACM Special Interest Group on Data Communication (SIGCOMM)* (2018).

[50] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. 2014. Scale-out NUMA. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2014).

[51] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen Rumble, Ryan Stutsman, and Stephen Yang. 2015. The RAMCloud Storage System. *ACM Trans. Comput. Syst.* 33, 3, Article 7 (Aug. 2015), 55 pages. https://doi.org/10.1145/2806887

[52] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. 2018. Floem: A Programming System for NIC-Accelerated Network Applications. *Symposium on Operating Systems Design and Implementation (OSDI)* (2018).

[53] C. Pinto, D. Syrivelis, M. Gazzetti, P. Koutsovasilis, A. Reale, K. Katrinis, and H. P. Hofstee. 2020. 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)*. 868–880. https://doi.org/10.1109/MICRO50266.2020.00075

[54] Arash Pourhabibi, Siddharth Gupta, Hussein Kassir, Mark Sutherland, Zilu Tian, Mario Paulo Drumond, Babak Falsafi, and Christoph Koch. 2020. Optimus Prime: Accelerating Data Transformation in Servers. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2020).

[55] Akshitha Sriraman and Thomas F. Wenisch. 2018. µSuite: A Benchmark Suite for Microservices. *Int'l Symp. on Workload Characterization (IISWC)* (2018).

[56] Akshitha Sriraman and Thomas F. Wenisch. 2018. µTune: Auto-Tuned Threading for OLDI Microservices. *USENIX Symp. on Operating Systems Design and Implementation (OSDI)* (2018).

[57] Patrick Stuedi, Animesh Trivedi, Bernard Metzler, and Jonas Pfefferle. 2014. DaRPC: Data Center RPC. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SOCC '14)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/2670979.2670994

[58] S. Sur, Lei Chai, Hyun-Wook Jin, and D. K. Panda. 2006. Shared receive queue based scalable MPI design for InfiniBand clusters. In *Proceedings 20th IEEE International Parallel Distributed Processing Symposium*. 10 pp.–. https://doi.org/10.1109/IPDPS.2006.1639336

[59] Mark Sutherland, Siddharth Gupta, Babak Falsafi, Virendra Marathe, Dionisios Pnevmatikatos, and Alexandros Daglis. 2020. The NeBuLa RPC-Optimized Architecture. *Int'l Symp. on Computer Architecture (ISCA)* (2020).

[60] Yanqi Zhang, Weizhe Hua, Zhuangzhuang Zhou, Ed Suh, and Christina Delimitrou. 2021. Sinan: Data-Driven Resource Management for Interactive Microservices. In *Proceedings of the Twenty Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.