Aquila: A unified, low-latency fabric for datacenter networks

Dan Gibson, Hema Hariharan, Eric Lance, Moray McLaren, Behnam Montazeri, Arjun Singh, Stephen Wang, Hassan M. G. Wassel, Zhehua Wu, Sunghwan Yoo, Raghuraman Balasubramanian, Prashant Chandra, Michael Cutforth, Peter Cuy, David Decotigny, Rakesh Gautam, Alex Iriza, Milo M. K. Martin, Rick Roy, Zuowei Shen, Ming Tan, Ye Tang, Monica Wong-Chan, Joe Zbiciak, Amin Vahdat
Google Inc.
aquila-nsdi2022@google.com

Abstract

Datacenter workloads have evolved from the data intensive, loosely-coupled workloads of the past decade to more tightly coupled ones, wherein ultra-low latency communication is essential for resource disaggregation over the network and to enable emerging programming models.

We introduce *Aquila*, an experimental datacenter network fabric built with ultra-low latency support as a first-class design goal, while also supporting traditional datacenter traffic. Aquila uses a new Layer 2 cell-based protocol, GNet, an integrated switch, and a custom ASIC with low-latency Remote Memory Access (RMA) capabilities co-designed with GNet. We demonstrate that Aquila is able to achieve under 40 μs tail fabric Round Trip Time (RTT) for IP traffic and sub-10 μs RMA execution time across hundreds of host machines, even in the presence of background throughput-oriented IP traffic. This translates to more than 5x reduction in tail latency for a production quality key-value store running on a prototype Aquila network.

1 INTRODUCTION

There has been tremendous progress in datacenter networking over the past decade, with fundamental advances in the control plane [18,27,44,49], the rise of commodity silicon arranged in non-blocking topologies [4, 23, 36, 49], network management and verification [7, 8, 29, 41], and highly available network design techniques [21]. Taken together, the community is now in a place where cost-effective, easy-to-manage, and scalable network designs and deployments are becoming common in industry. Plentiful network bandwidth at the scale of clusters of tens of thousands servers [49] can be leveraged for large-scale hyperscalers and the services they host.

However, all of these advances come while assuming TCP-based congestion control and Ethernet Layer 2 protocols. This Layer 2-4 stack has been incredibly robust and resilient through many decades of deployment and incremental evolution. However, we are seeing a new impasse in the datacenter [12] where advances in distributed computing are increasingly limited by the lack of performance predictability and isolation in multi-tenant datacenter networks. Two to three orders of magnitude performance difference [15] in what network fabric designers aim for and what applications can expect and program to is not uncommon, severely limiting the pace of innovation in higher-level cluster-based distributed systems.

Such concerns are amplified when considering the state of supercomputing/HPC clusters [17] and emerging machine learning pods [22, 28] where individual applications benefit from low-latency RDMA [16, 51], collective operations [46], and tightly integrated compute and communication capabilities. The key differences in these more specialized settings relative to production datacenter environments include: i) the ability to assume single tenant deployments or at least space sharing rather than time sharing; ii) reduced concerns around failure handling; and iii) a willingness to take on backward incompatible network technologies including wire formats.

Recent research efforts into disaggregated rack-scale architectures [13, 34] further highlight some of these challenges: can the same NICs and switches supporting host-to-host communication across the wide area support, for example, SSD and GPU devices at a much smaller radius? Is the disaggregation network necessarily a separate dedicated fabric or can it be multiplexed with TCP/IP traffic destined to remote hosts potentially 100ms or more away? While there is some appeal to running a second (or third) network dedicated for an individual use case, the control and, as importantly, the management overhead of each network introduces a cyclic dependency where the second network is not worthwhile relative to the status quo until the underlying technology is proven/mature. However, there is no opportunity to iterate on the alternate technology because doing so is cost and complexity negative for a number of generations into the future because applications would have to evolve substantially before demonstrating end-to-end wins on the new hardware.

The need for backward compatibility combined with challenges in deploying niche "bag on the side" networks threatens a new ossification in datacenter networking and distributed systems where we are left with programming to the lowest common denominator of TCP transports and commodity Ethernet switches with associated latency, CPU efficiency, and isolation limitations.

In this paper, we present a first exploration of an alternative tightly-coupled (or *Clique*-based) datacenter architecture, *Aquila*, a hardware implementation supporting predictable, high-bandwidth, and ultra-low latency communication. In our approach, datacenter networks consist of dozens of Cliques, each hosting approximately 1-2k network ports. Cliques interoperate with one another at the datacenter interface (e.g., the spine layer of existing Clos-based datacenter networks) through standard Ethernet and IP. However, within a Clique, any transport and Layer 2 network protocol may be deployed. Applications that fit within the boundaries of an individual Clique can assume Clique-local capability, including robust RDMA, predictable low-latency communication, device disaggregation, support for ML aggregation primitives, etc. We assume IP-based transport for communication between Cliques, which means that any intra-Clique communication primitives and innovations must live alongside standard transports. Cliques then become the unit of deployment, innovation, and homogeneity, allowing for incremental, backward-compatible deployment into existing datacenters. A Clique is also sufficiently large to host all but the largest of individual distributed systems, especially as we move to hundreds of compute cores per server.

Aquila, our first Clique implementation based around a custom in-house ASIC and communication software, consists of a cell-switched non-Ethernet substrate, GNet.

- Aquila networks are built from individual silicon components that serve as both NIC and a portion of the traditional Top of Rack (ToR) switch; each *ToR-in-NIC* (TiN) chip attaches to hosts and directly to other TiN chips to realize a cost-effective network built from a single, replicated silicon component, rather than distinct NIC and switch silicon components from separate vendors.
- GNet provides the illusion of Ethernet to hosts within Aquila, as well as to non-Aquila networking components outside the scope of the Aquila Clique, by terminating Ethernet at the Aquila network boundary and tunneling traffic across a fully-custom, self-defending, near-lossless L2 substrate.
- Aquila further reduces cost by realizing a direct-network rather than an indirect (Clos) topology. To fully unlock the capabilities of its Dragonfly topology [30], and freed from the de facto constraints imposed by Ethernet, Aquila leverages adaptive routing to deliver full point-to-point bandwidth between host-pairs by leveraging multiple non-minimal paths.
- Aquila delivers data in small chunks called *cells*, rather than packets, thereby optimizing for latency of small exchanges like those used by distributed systems built on RDMA and similar technologies [51]. Its extremely tight integration between NIC and network allows for ultra-low RMA-read capability (4us median) between the memory systems of up to 1152 hosts.

Aquila's design departs from traditional Ethernet fabrics in several ways: i) links use credit-based flow-control; ii) switch buffering is shallow; and iii) solicitation bounds end-to-end admission. Any one of these tenets in Ethernet would be problematic, but taken together, they form a cohesive design. For instance, flow-controlled near-lossless links can give rise to tree saturation, especially with shallow buffering, but end-to-end admission control bounds the size and spread of such trees, and ensures they are transient. Similarly, admission control breaks down when drops are likely, but link-level flow control makes drops very rare, and in turn enables the use of shallow buffering in switching elements, since overrun is not possible.

We present the detailed design, implementation, and evaluation of Aquila. Aquila is not the final word in Clique design; in fact, our first experience with the Aquila system suggests a number of areas for improvement in future generations. We hope, however, that the approach of bringing vertical integration including the host software stack, the NIC, and the switch along with a Clique-based datacenter architecture will enable new models of datacenter innovation along with new capabilities to distributed systems that can assume cutting edge rather than lowest common denominator communication and disaggregation capability within the boundary of thousands of servers and hundreds of thousands of cores.

2 OBJECTIVES AND OUR APPROACH

Aquila's design departures from Ethernet are grounded in a set of common objectives, described below. Taken individually, these design choices–e.g., flow control, custom Layer 2–would be hard to apply to an existing network incrementally. But in concert, Aquila's features realize a complete, performant design point.

Sustainable hardware development. To sustain the hardware development effort with a modest sized team, we chose to build a single chip with both NIC and switch functionality in the same silicon. Our fundamental insight and starting point was that a *medium-radix* switch could be incorporated into existing NIC silicon at modest additional cost and that a number of these resulting NIC/switch combinations called *ToR-in-NIC* (*TiN*) chips could be wired together via a copper backplane in a *pod*, an enclosure the size of a traditional Top of Rack (ToR) switch. Servers could then connect to the pod via PCIe for their NIC functionality. The TiN switch would provide connectivity to other servers in the same Clique via an optimized Layer 2 protocol, GNet, and to other servers in other Cliques via standard Ethernet. The inset in Figure 1 summarizes the major components of TiN.

Cost effective, non-blocking topology. For efficiency and low latency, we selected a direct connect topology, Dragonfly, a well-studied topology that minimizes the number of

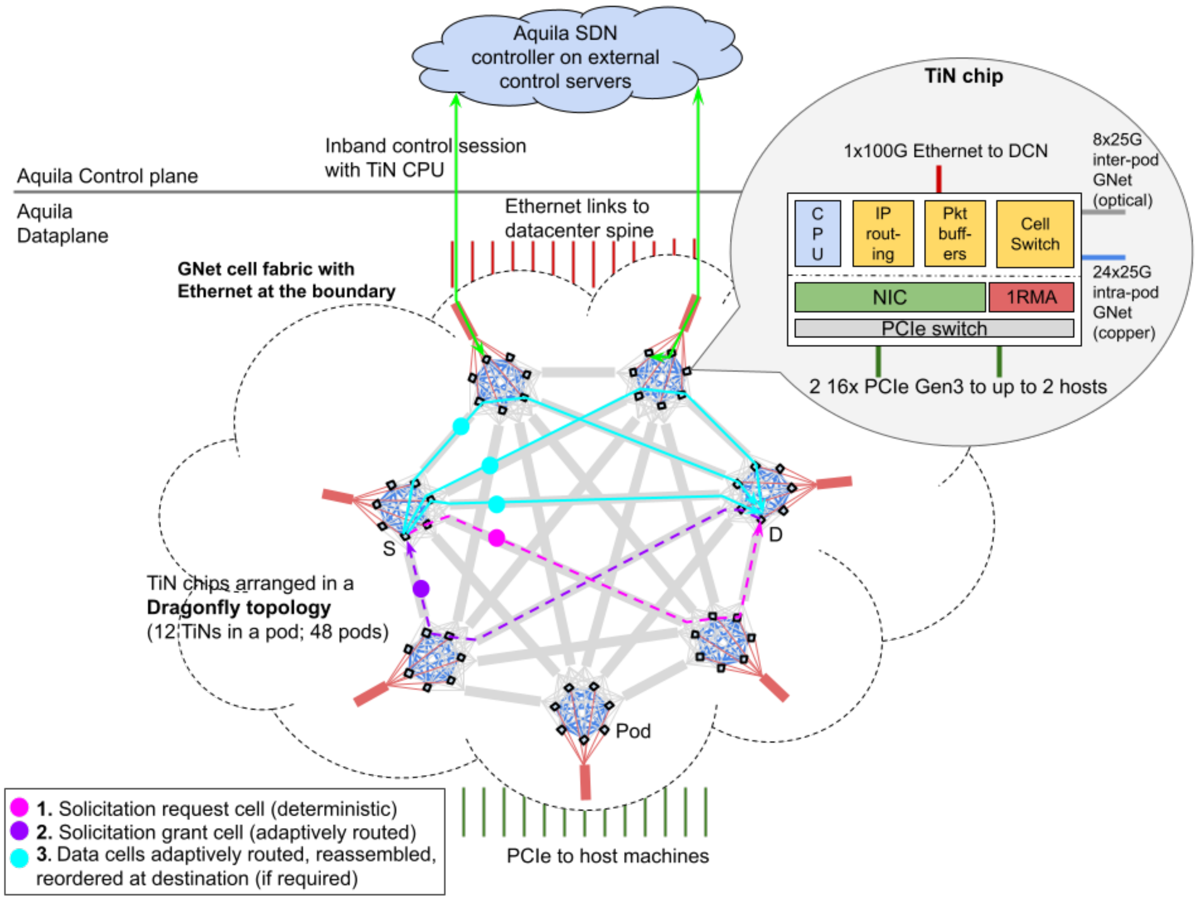
Aquila SDN controller on external control servers
TiN chip
Inband control session with TiN CPU
Aquila Control plane
Aquila Dataplane
Ethernet links to datacenter spine
1x100G Ethernet to DCN
8x25G inter-pod GNet (optical)
CPU
IP rout-ing
Pkt buff-ers
Cell Switch
24x25G intra-pod GNet (copper)
NIC
1RMA
PCIe switch
2 16x PCIe Gen3 to up to 2 hosts
GNet cell fabric with Ethernet at the boundary
S
D
TiN chips arranged in a Dragonfly topology (12 TiNs in a pod; 48 pods)
Pod
1. Solicitation request cell (deterministic)
2. Solicitation grant cell (adaptively routed)
3. Data cells adaptively routed, reassembled, reordered at destination (if required)
PCIe to host machines

Figure 1: Aquila Clique dataplane and control plane overview. The ToR-in-NIC (TiN) chip is expanded in the top right inset. TiN chips are arranged in a cell-switched Dragonfly network, connecting via Ethernet to the rest of the datacenter network's (DCN) spine switches and to host machines via PCIe. Conventional Ethernet/IP packets are split into cells at ingress after a round of solicitation per packet and reassembled and re-ordered at the fabric's egress. The co-designed 1RMA protocol injects cells directly into the cell network, extending memory accesses across the Clique. The Aquila SDN controller configures and manages TiN switches inband, via the DCN.

long optical links in the network while still providing non-blocking bandwidth for uniform random traffic patterns, with 2:1 over-subscription for worst-case adversarial traffic. Figure 1 illustrates a simplified Dragonfly topology where TiNs within a pod are fully connected in a mesh, and multiple pods are likewise connected all-to-all to form a tightly-coupled Clique network. The largest Aquila network supports 12 TiNs in a pod with 48 pods, serving up to 1152 host machines.

Combining NIC and ToR into the single TiN chip was a less costly path to innovation than separate NIC and switch ASIC programs, and a design realized from a common single component was intended to streamline inventory management for Aquila. Further, we implemented an optional capability to allow pairs of host machines to share a single TiN, halving the normalized cost of ownership for networking per host, trading off reduced sustained bandwidth provisioning per machine.

Ultra low-latency network. To optimize for ultra-low latency, under load and in the tail, Aquila implements cell-based communication with shallow buffering for cells within the network, flow controlled links for near lossless cell delivery, and hardware adaptive routing to react in nanoseconds to link failures and to keep the network load balanced even at high loads. To ensure recipients are not overwhelmed, Aquila implements end-to-end solicitation for each packet at ingress, which guarantees that resources are available at the destination TiN before the packet can be split into cells and transmitted from the source TiN. We built these latency-guarding features into Aquila's Layer 2 protocol, GNet. As depicted in Figure 1, while the Aquila network fabric presents an Ethernet packet interface at its boundary, Aquila tunnels conventional Ethernet/IP packets over GNet, disassembled at ingress and reassembled and re-ordered at the egress of the Clique.

Unified fabric for legacy traffic and RMA/memory disaggregation. Aquila unifies low-latency communication primitives (RMA) alongside commodity primitives (IP) in a common fabric, to address the growing diversity of datacenter workloads [5, 42, 45]. A fabric delivering both high-performance and legacy connectivity avoids the pitfalls of a *bag-on-the-side* network and secondary NICs, reducing the cost of ownership and the toil related to the life cycle management of two separate networks. Managing a single network for availability, security, monitoring and upgrades is challenging enough–managing separate networks for individual use cases introduces an extraordinarily high bar in any cost/benefit analysis. For efficient remote memory access and memory disaggregation alongside traditional protocols, we co-designed a Remote Memory Access protocol, 1RMA [51], to extend memory access across the Aquila Clique directly on GNet, instead of layering on top of IP.

Co-existing within the larger Clos-based software-defined datacenter network ecosystem. Typical datacenter networks [49] are based on a scalable Clos topology where aggregation blocks are connected via a spine switching layer; Aquila is designed to integrate into such a network via its Ethernet ports. A hierarchical Software Defined Network (SDN) control plane with a modular, micro-service architecture [18] manages and controls the various networking blocks within the datacenter.

Figure 2 describes the integration of Aquila in the broader datacenter network's dataplane and control plane ecosystem. The Aquila network block connects to the datacenter's spine switching layer via Ethernet links, akin to other aggregation blocks. The modular architecture of the datacenter network realizes a hybrid topology, i.e., a Dragonfly network integrated as a block within a larger Clos topology, a first of its kind to the best of our knowledge.

For the control plane, we adapted an SDN controller to configure, manage and program TiNs inband via a thin on-box firmware running on the TiN CPU (Figure 1). The Aquila SDN controller, similar to the SDN controllers of other aggregation blocks, interacts with each of four central Inter-Block Routing Controllers (IBR-C) (Figure 2) to enable communication with other aggregation blocks as well as with networks external to the datacenter.

Cliques as the basis for hosting tightly-coupled applications. To exploit the tightly coupled, low latency communication enabled by the Aquila Clique, we adapted the job scheduler [53] to be aware of Clique locality. High bandwidth or latency sensitive jobs could optionally be scheduled on host machines within a Clique, while other jobs could still be bin-packed across blocks, regardless of locality.

3 HARDWARE DESIGN

In this section we relate how the key design goals drove the hardware design. In summary:

- **Low latency** objectives drove the selection of a shallow-buffered cell-switched GNet fabric. §3.1 details the design of the GNet switch and link-level protocol.
- **Cost-effectiveness** goals led to the choice of an integrated switch and NIC chip, TiN, as well as a direct topology such as the Dragonfly. §3.2 outlines the rationale for selecting the Dragonfly and the impact of this choice on the design.
- **Shared fabric for both IP traffic and low-latency RMA.** §3.3 describes how IP packets traverse the GNet fabric, and §3.4 details the co-design aspects of the 1RMA protocol with the GNet fabric.

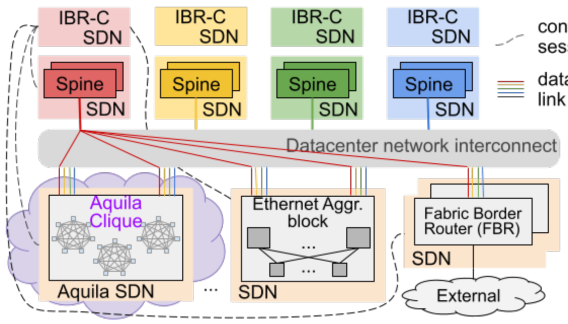

Figure 2: Aquila Clique integrated into the broader datacenter network and SDN ecosystem co-existing with other Ethernet aggregation blocks. Topologically, the Aquila block connects to the Clos-based datacenter spines akin to the Ethernet based blocks. In the control plane, the Aquila SDN controller, similar to controllers of other blocks, interacts with each of the four *sharded* Inter Block Routing controllers (IBR-C) to enable cross-block routing.

3.1 GNet Switch and Links

The cell switch. The switching capability of the TiN chip is provided by a 50-port cell switch optimized for low latency. The maximum cell size of 160 bytes was chosen to keep the serialization latency on 25G links small (~50ns). 32 ports are external-facing GNet ports (of which 24 are pod-local and 8 are inter-pod ports). The remaining 18 ports are intra-chip, for cells transmitted and received by the various traffic endpoints (e.g., IP and 1RMA). The fall through latency of the core cell switch is 20ns and the total per hop latency without Forward Error Correction (FEC) is 40ns. GNet links support 32 Virtual Channels (VCs [14]) - FIFO queues used for deadlock-avoidance and QoS. VCs are used for deadlock-free routing, for differentiation between classes of service, and to separate solicited and unsolicited traffic. A centralized arbiter implements a variant of the iSLIP arbitration protocol [38], supporting one arbitration request per VC per port, ensuring that no VC or port is starved of throughput. To support variable cell sizes, we modified iSLIP such that the ingress and egress ports communicate a "busy" signal to the crossbar arbiter. A "busy" indicates that the ports are transferring a cell across the crossbar. The arbiter takes this into account when it evaluates pending requests for the next request-grant-accept cycle. Quality of Service (QoS) between VCs is implemented in the output buffer and supports both weighted round robin and strict priority. Each VC has its own input FIFO space protected by a reliable credit mechanism, similar to that used in PCI Express. A shared buffer, shared credit scheme was considered to save memory, but for the relatively short links required for Aquila the simplicity and complete QoS isolation of independent FIFOs was preferred.

GNet link level protocol. GNet links support cells between 16 bytes and 160 bytes in size, with frequent reverse flow control traffic. The use of variable length cells gives very high protocol efficiency (e.g., 1RMA requests are small) on the wire even after the additional control traffic for admission control. Every GNet cell has a common routing header of 8 bytes that contains the 16 bit source and destination GNet addresses, the cell length and type, the VC, a decrementing hop count, an 8 bit header CRC and a Trace Enable bit. To enable efficient transmission of GNet cells, a custom 66/64 bit Physical Coding Sublayer (PCS) was developed that minimizes the cell delineation overhead and allows the reverse flow control traffic to be sent as very compact ordered sets. Control ordered sets are used for: (1) Start of cell delineation, (2) Flow control, (3) TimeSync (§B), (4) Management ordered sets (MOS), and (5) Phy up/down control and fault detection.

3.2 The Dragonfly cell fabric

We selected the Dragonfly topology to manage the cost of optical links, shown in Figure 1. The Dragonfly cell fabric is implemented using two types of GNet links: 24 *local* links per TiN that fully connect TiNs within the pod, and 8 *global* links per TiN that connect between pods, up to 100m apart on the datacenter floor. The local links are implemented as single-lane, 28 Gbps copper backplane connections. The global links are optical and use specially developed low cost GNet optical modules. Noise on global links is mitigated with FEC, incurring a 30ns per-hop latency penalty, and a 6% bandwidth overhead. Local links operate without FEC for the lowest latency at acceptable margins. Both local and global links ultimately implement the same link level protocol. Due to the hierarchical nature of the Dragonfly topology, GNet addresses have three components: *pod id*, *TiN id* and *endpoint id*. Endpoints represent protocol engines (IP, 1RMA, and CPU) detailed later.

Deadlock avoidance. We implement deadlock avoidance in our Dragonfly topology using a combination of turn rules and VCs. With our budget of 32 VCs, it is desirable to minimize the number of VCs used for deadlock avoidance. In the implemented routing scheme, we employ turn rules similar to the *parity-sign* approach in [20] within a pod for deadlock-free intra-pod routing. The VC is incremented when moving from a global link to a local link [30], requiring a total of 3 VCs used for deadlock avoidance in the worst fault-free route, that of a non-minimal route via an intermediate pod. Accounting for 10 traffic classes, each with 3 routing VCs, a further two VCs are available as *escape VCs* in certain dynamic failure avoidance scenarios.

Adaptive routing. The majority of traffic routes adaptively to achieve both high throughput and the lowest latency on Aquila's Dragonfly network. TiN implements locally adaptive routing [30, 48], a scheme that makes adaptive routing decisions based on available information at a GNet switch, in particular, the per-VC output queue lengths at each port. These queue depths reflect nearby congestion because of GNet's link-level flow control and shallow buffering. Link failures manifest similarly, which also allows the adaptive routing algorithm to route around failed links until the SDN routing engine removes the entries for links which have lost connectivity.

The adaptive routing implementation selects two minimal routes at random from eight supplied by the routing tables, and also considers three non-minimal routes from 24 non-minimal route candidates. The five candidate routes are evaluated using a weighted comparison that favors the minimal routes. Random choices (rather than 24-way comparison) allow us to avoid flocking, having coordinated adaptive routing decisions, and moving congestion from one place to another [40]. Other routing modes are enabled by constraining the routing to minimal routes only, or by forcing deterministic choice of route using a hash of the source and destination addresses. These constraints yield Aquila's four principal routing modes: Fully Adaptive, Minimal Adaptive, Deterministic and Minimal Deterministic. The deterministic routing modes are used for cell types requiring ordering. The cell switch uses table-based routing because of the need to handle failures and upgrades using SDN routing described in §4.2, as well as flexibility for other topologies.

3.3 IP Traffic

Host IP traffic is sent and received by a conventional 100 Gbps NIC capable of supporting multi-host operation for up to two independent hosts. The option of multi-host capability was considered important in order to give a degree of flexibility in bandwidth per machine allocation. There are two other sources of IP traffic on the TiN chip: the 100 Gbps external Ethernet port for connectivity outside the Aquila fabric, and a low bandwidth port to the embedded management processor. Traffic from all IP sources is handled in the same way: the packet processing pipeline performs IP routing and cellification, i.e., splitting the IP packet into GNet cells and traversing them to the final destination, using Aquila's IP over GNet protocol.

Packet processing logic. Each IP packet passing over the GNet fabric goes through the input packet processing and output packet processing blocks once only. Effectively, the entire GNet Clique acts as a single stage IP packet switch. The input packet processing pipeline handles:

- L3 to GNet L2 address translation (either one-to-one or WCMP [55]);
- Selectively punting some packets to the embedded control processor;
- Input buffer QoS.

L2 Ethernet MAC addresses are stripped from inbound packets after processing; packet transfer over the GNet cell network is for IPv4/IPv6 only. Non-IP packets such as ARP may be either encapsulated or punted to the embedded control processor, consistent with the requirements of our SDN control plane (§4).

IP over GNet protocol. IP traffic traverses Aquila by means of the GNet upper layer protocol, shown in Figure 3. Each IP packet sent over the cell fabric issues a *Request To Send* (RTS), and awaits a *Clear to Send* (CTS) handshake before any data is transmitted. These are sent as 16 byte GNet cells to minimize the bandwidth overhead. The handshake protocol performs three functions:

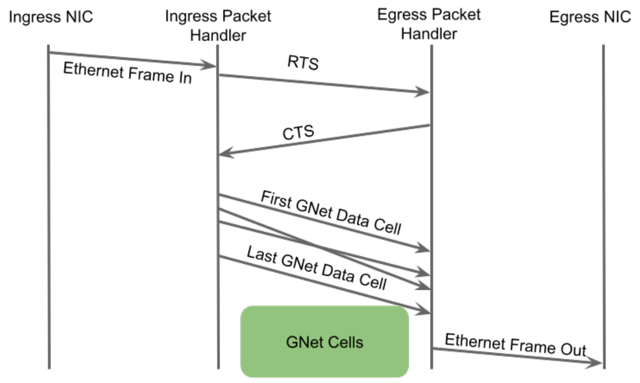

Figure 3: Cellification: IP Packets are split into multiple GNet data cells that are only admitted into the GNet fabric when the ingress TiN receives a CTS. In the egress packet handler, cells are reassembled into packets, respecting their original transmission order, and sent to the NIC (all within the TiN chip).

- It implements solicitation for IP packets by only allowing data cells onto the GNet network when the destination end point has signalled it has sufficient input bandwidth and buffer space to receive them.
- It allocates hardware resources at the destination, e.g., cell to packet reassembly buffers, before any data cells are transmitted so that there is always a guaranteed reassembly space.
- RTS arrival defines inter-packet order. While data cells route adaptively and potentially arrive out of order at their destination, RTS ordering ensures that original transmission order can be reconstructed at the receiving side.

Packets which have passed through the packet processing pipeline and have a valid GNet L2 address are stored in the packet ingress buffers. An RTS is generated immediately; in fact, for long packets the RTS can be issued before the whole packet is received. The RTS itself consists of 8 bytes of routing header and a further 8 bytes of payload that includes the IP packet length, Class of Service (CoS), the packet's location in the ingress buffer, and an indicator of ingress buffer usage. Compactness is important because RTS cells are unsolicited and can still lead to incast. However, considering that an average packet is >1Kbytes, an incast of RTS cells represents a reduction of incast volume in the network by a factor of 64.

RTS cells are carried on their own VCs, allowing them to be sent at high priority and also maintain isolation between solicited and unsolicited cells. RTS VCs are routed deterministically over the GNet fabric, using a path selected by a hash of the flow-invariant fields of the cell, ensuring that the RTS cells for a given IP flow are received in the order they were sent. The RTS cells are received into FIFO queues at the packet egress. Packet data transfer is initiated by the egress-side by sending a CTS back to the appropriate ingress port. Along with the 8-byte routing header, the CTS carries a pointer to the packet in the ingress buffer (copied from the RTS) and a pointer to the allocated location in the egress cell-to-packet reassembly buffer. CTS cells are issued by the CTS scheduler, which tracks the availability of egress reassembly buffer capacity, only issuing a CTS when there is space available to reassemble cells into packets.

When a CTS is received back at the packet source, the packet in question is pulled from the ingress buffer as a series of data-only cells, which are then transmitted across the fabric. Data cells can take many different routes (adaptively) through the fabric, and data cells may arrive in any order at the final destination. Cells are reassembled into packets in the egress buffer at the destination. The sizing of the egress buffer is determined by the bandwidth delay product of the output port bandwidth and the cell fabric round trip delay, plus an allowance for packet reordering delays. Packets do not experience significant queuing in the egress buffers, which are primarily for reassembly, so the egress buffers are significantly smaller than the ingress buffers.

In order to maintain packet order within flows, when a CTS is issued by the scheduler, the packet descriptor is registered with packet reordering logic respecting RTS arrival order. A packet is transmittable at egress after receipt of all its data cells, but transmittable packets are held until all packets in the same flow that were ahead of it in CTS issue order have been successfully forwarded to the NIC.

A significant benefit of the RTS/CTS scheme is that the RTS queues have a local view of all the requested packet demand for that destination port from the entire GNet fabric, while the packet *data* remains queued in the ingress buffers. In the presence of severe incast, packets can be discarded while conserving fabric bandwidth, i.e., without packet data traversing the cell fabric. The egress side can choose to drop a packet by issuing a variant of CTS (a *Clear To Drop*, CTD), which pulls the packet from the ingress buffer and discards it. A CTD is sent when an RTS is received at an RTS queue whose depth exceeds a given threshold. The RTS queue's depth also provides the signal for Explicit Congestion Notification (ECN) marking; if the RTS queue exceeds the marking threshold when the packet has been reassembled and is ready to send, ECN is applied.

QoS Support for IP. There are separate RTS queues for each independent port and class of service, with a total of 32 RTS queues supporting eight CoS on four independent ports - one port for the external Ethernet MAC, two for the dual-host NIC, and one for the control processor. CTS cells are issued by the CTS scheduler at the packet's destination which allocates bandwidth between the 32 RTS queues, implementing per-IP-packet QoS between the respective queues. The CTS scheduler may throttle traffic into the egress buffers by limiting CTS issues according to a window of outstanding packet fetches, which can be adjusted to minimize the queuing of data cells within the cell fabric. The scheduler does not attempt to implement bandwidth fairness between sources since all the sources to a given destination port share the same RTS queue.

3.4 1RMA

To deliver the low-latency capabilities of the Aquila Clique directly to distributed systems programmers, we built an implementation of 1RMA [51] into the TiN chip. 1RMA is an RMA protocol that offers unordered, segmented, solicited remote memory access primitives (read, write, and atomics) to on-host software—tenets that match precisely those of GNet packet transfer governed by RTS/CTS.

Such alignment is not merely coincidental; we co-designed Aquila and 1RMA's GNet-based protocol. Rather than simply layering the 1RMA protocol messages above the packet layer, we instead express 1RMA protocol exchanges as first class cell types in GNet—alongside RTS and CTS, rather than atop—and ensure that they obey similar end-to-end solicitation rules as they share the Aquila fabric. The advantage of co-design is significant latency savings: while a UDP/IP or TCP/IP round-trip on Aquila incurs *six* GNet half-round-trips on its critical path (RTS, CTS, data, in each direction), a 1RMA read operation incurs only *two*, shaving precious microseconds from user-facing latency.

We realized protocol co-design by encoding 1RMA *read requests* entirely within GNet framing. Fundamentally, read requests initiate data transfer from receivers to senders, i.e., such requests intrinsically already are solicitations, expressed at the transport layer. GNet also builds on solicitation, but at the L2 layer. The key insight is to express both the GNet (L2) and 1RMA (L4) solicitation behaviors in a single cell type, *Req*. Since Req cells solicit data movement in the reverse direction, GNet handles Req similarly to CTS; the main differences arise from cell size, as Req fully encodes a read request (host address, memory identifiers, HMAC, etc.), yielding a cell 3x larger than CTS at 48B. Req is otherwise behaviorally similar to CTS, in that it can be freely reordered without violating assumptions of the protocol layer above. Because 1RMA is highly tolerant of out-of-order delivery, Req is intrinsically compatible with Aquila's adaptive routing.

We also leverage 1RMA's close coupling to host-facing PCIe to encode response cells, *Resp*. 1RMA NICs send each individual PCIe read completion payload as a distinct protocol response, a hardware simplification that avoids response coalescing logic, buffering, and overheads in the NIC. To facilitate this behavior in GNet, Resp cells are sized to handle the most common PCIe completion sizes we observe from the host root complexes. Like Req, Resp can be freely reordered and routed adaptively, and the initiating 1RMA NIC lands the individual response segments in arrival order in destination host memory, since there is no need to restore overall inter- or intra-request response ordering.

Lastly, to isolate latency-critical 1RMA traffic from less sensitive IP flows, we map roughly half of GNet's virtual channels to carry low-latency protocol messages, which 1RMA shares with low-latency IP traffic flows. Because IP traffic is cellified, 1RMA responses do not queue behind bulk transfers from competing flows. In all, 1RMA on Aquila delivers near-flat lookup latency—even under load from conventional traffic—to approximately 864TB of DRAM inside of 4us end-to-end. Aquila traversal accounts for a mere 2.5-3us; the remaining time is attributable to PCIe latency contributions.

3.5 Embedded control processor

The TiN chip has an embedded control processor (ECP) to handle all switch side control and monitoring actions. Cost of silicon exerts pressure to make the ECP as simple as possible, as it is replicated in each TiN chip. Where a typical control processor for a ToR might be a multicore, 64-bit processor with 8-16 GB of memory, TiN's ECP is a 32-bit ARM Cortex M7 processor with a mere 2 MB of SRAM.

In order to bootstrap the embedded control processor before the GNet logic has been fully initialized (§4.4), a low bandwidth but reliable in-band control path is implemented over the GNet fabric using the management ordered set (MOS). Each MOS 64 bit word allows 6 bytes of data to be transferred between directly connected TiN chips, irrespective of whether the GNet link layer is up. We layer a robust packet implementation, PMOS, above the MOS primitive to carry debug and bootstrap traffic.

3.6 Putting it all together

Figure 4 plots the overall structure of the TiN chip:

- The cell switch (the building block for the cell fabric);
- A conventional IP host interface (NIC);
- An external-facing Ethernet MAC for connectivity to outside networks;
- A 1RMA host interface that supports direct protocols across the cell fabric;
- IP packet-to-cell (ingress) and cell-to-IP packet (egress) logic;
- The embedded control processor, acting as the local agent for the SDN control software.

The device has the following interfaces:

- Two x16 PCIe gen 3 interfaces giving 256 Gbps connectivity to one host or 128 Gbps to each of two hosts;
- Thirty-two 28 Gbps, single-lane GNet links used to construct the low latency cell fabric;
- A single 100 Gbps Ethernet interface which connects to the wider datacenter network (DCN).

Approximately 50% of the TiN silicon area is used to implement host interface or NIC functions, and 50% for switching functions.

4 SOFTWARE-DEFINED NETWORK

As alluded to in §3, the integration of switch and NIC in a single chip leads to substantial replication of the management subsystem across all TiNs in an Aquila Clique. To keep the Aquila Clique cost-effective, the management subsystem for a TiN ASIC was kept simple – a 32-bit ARM Cortex M7 processor with a modest 2MB of SRAM and no dedicated management Ethernet port. Consequently, much of the routing

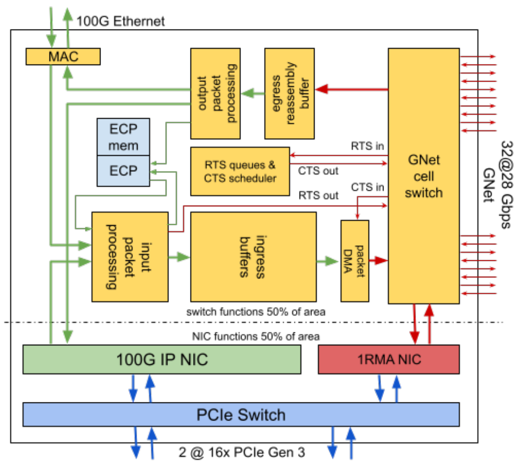

Figure 4: Aquila Chip Architecture showing GNet, IP, Embedded Control Processor (ECP), 1RMA and PCIe components.

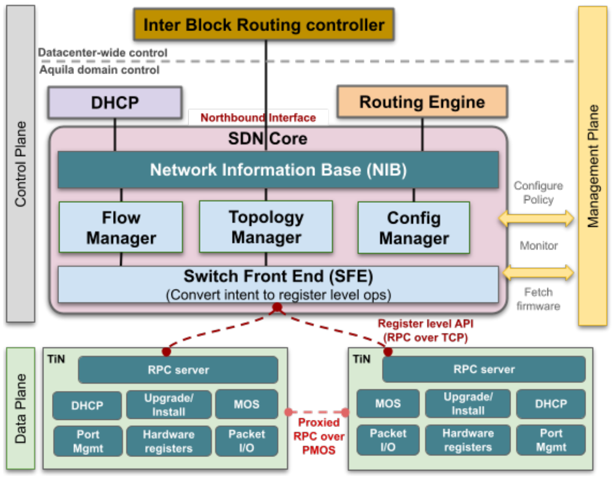

Figure 5: Overview of Aquila's SDN and firmware architecture.

computation and state needed to be offloaded from the switch to a logically centralized distributed controller which had to orchestrate the bootstrap, management and control of the fabric in-band. In this section, we describe how Aquila's software-defined network (SDN) control plane, along with its simplified firmware, was able to address the challenges of controlling and managing an Aquila Clique, specifically:

- Explosion of flow state in the SDN (§4.2).
- Switch firmware with constrained CPU/memory (§4.3).
- In-band bootstrap of the whole network (§4.4).

4.1 Control Architecture Overview

The Aquila controller is built on top of an SDN controller platform [18], a modular SDN control plane comprised of micro-services, and a central publisher/subscriber database called the Network Information Base (NIB). Multiple applications form an Aquila SDN constellation with redundant instances of each application deployed on separate control servers. The top half of Figure 5 details the Aquila SDN controller applications.

The routing application for the controller, Routing Engine (RE), computes the routing solution for the Aquila network block in reaction to changes of topology states and external reachability. RE writes the solution in the form of flows and groups similar to OpenFlow [39] to the NIB in sequenced batches for hit-less routing state transition. Separately, the Inter-Block Routing controller (IBR), an application in a data center-wide SDN control domain, computes the routing solution for traffic between various network blocks and provides Aquila's RE with the egress paths to reach destinations external to the Aquila Clique.

On receiving routing updates from the NIB, Flow Manager (FM) sorts the flow and group programming operations. For instance, a flow is installed only after its referenced group is installed for hit-less transition, before sending them to the Switch Front-End application (SFE) via RPC. SFE programs the flows and groups to TiN switches converting between flows/groups and hardware register values and completes the RPC with the programming results. Then FM writes the results back to the NIB for RE to consume.

4.2 Handling routing state

The large number of GNet endpoints in the fabric and the per-port GNet routing table in the TiN switch result in much larger routing state than non-Aquila blocks, which increases both CPU and memory demand in the SDN system. Aquila routing introduced scaling challenges for both IP and GNet flows.

IP flows. A network comprised of 1152 hosts and 576 management CPUs, addressable via both IPv4 and IPv6, calls for approximately 1.9 million flows, each with a single output port. Leveraging the observation that all of these flows are from a small number of subnets, we introduced a new *indexed* group representation, where the index of a port in the group corresponds to the same index in the subnet, which in turn reduces the number of flows by a factor of 576 (the number of TiNs in a fabric).

GNet flows. As seen in §3, flow controlled GNet requires per-input port, per-virtual channel flows which leads to an explosion in state for a switch with 50 ports. A naive implementation leads to almost 5 million flows. To accommodate such a large scale, we exploited the significant similarity in the routes. For example, all terminal ports described in §3.1 use the same route, and are represented only once in the NIB. Similarly, the deadlock avoidance turn rules define a similar role for each intra-pod port in the TiN chip. Further, all inter-pod ports behave the same. We introduced six *port classes* – denoting equivalence classes of ports with respect to routing rules – reducing the number of flows to approximately 700k. GNet flows use port-classes as both matching fields and output actions. On receiving a GNet flow using a port-class, SFE expands it to flows targeting each member port's GNet flow table, and then prunes improper member ports from the output, e.g., to avoid sending traffic back to the source. The resulting flows are then programmed in the switch.

Even with these optimizations in place, the rest of the SDN system needed more modifications:

- Despite the port-class optimization, the number of flows in the NIB was still about 10x more than non-Aquila network blocks. To compensate for the memory increase, the NIB's pub-sub interface was changed to keep state in compressed format and decompressed only when necessary.
- The SRAM available in the TiN switch is not large enough to hold a snapshot of all routing state. SFE has the capability to rate limit the hardware programming operations to avoid the memory on the switch from overflowing. The RPC interface between SFE and switches is designed in such a way that the largest RPC can fit in memory and only one outstanding RPC is allowed at a time.

4.3 Switch Firmware with limited state

The switch firmware (see lower half of Figure 5) runs on an ARM Cortex M7 CPU integrated into the TiN switch chip. Due to physical size and cost limitations, the firmware has only 2MB of on chip SRAM available. Therefore, it is built on the FreeRTOS [1] and lwIP [2] open source libraries to fit within the space constraints. The firmware is implemented in approximately 100k lines of C and C++.

We explicitly decided that the firmware is not responsible for fully configuring the TiN chip. At power on, the firmware brings up the GNet and Ethernet links and attempts DHCP over Ethernet. This enables the controller to connect early during initialization and finish the necessary configuration to allow the TiN chip to start passing traffic (for details see §4.4).

The programming API exposed by the firmware is low-level and allows the SDN controller to directly access hardware registers. The API is generated from the hardware register description and permits the SDN controller code to use symbolic names of the chip registers for convenience. Statistics and counters from the TiN chip can also be reported using the low level API. The SDN controller is able to configure a set of registers that should be periodically reported by the firmware. One of the programming API sets up ARP/NDv6 responses in reaction to requests from the attached machines so that the IP-to-MAC resolution could function properly even if the firmware loses connection to the SDN controller.

The firmware supports *Non-Stop Forwarding* (NSF) reboots to minimize disruption caused by upgrades and unexpected software errors. During reboot the firmware avoids changing any configuration that might impact traffic. Since the inband connectivity is not disrupted, the controller is able to quickly reconnect after a reboot without going through the bootstrap process. The implementation of NSF reboot was simplified due to the register level API since there is no need to save and restore state information, because the TiN chip maintains all the controller visible state during reboot.

While the firmware itself is stateless, the TiN chip and SDN controller are not. After any loss of connection between the firmware and SDN controller a process of reconciliation has to be initiated to resolve any differences between the hardware registers and the SDN controller intent. These differences can occur if any commands were lost when the connection failed.

4.4 In-band Control and Bootstrap

A key challenge in Aquila's SDN control was that the control channel from the SDN controller to the Aquila switches is in-band. This means that the controller needs to communicate with the management CPU of a TiN before it can program the routing tables of the TiN. During bootstrap, the controller sets up TCP connections in-band over the datacenter network to all TiNs in the Clique in iterative "waves", configuring and programming routing tables as it gains control of TiNs in each subsequent wave.

Figure 6 shows k Aquila pods connected via intra-pod copper GNet links as well as inter-pod optical GNet links. Some TiNs (e.g., TiN 1, TiN 3 in Pod 1 and Pod k) are also connected to the spine layer of the datacenter via Ethernet datacenter network (DCN) links. We refer to these TiNs as *DCN-connected*. The Aquila SDN controller—running on external control servers—is initially reachable only over the DCN links. The TiN firmware sends DHCP discover messages over the DCN links if available. These DHCP messages are relayed by the spine switches to the DHCP server, which then assigns an IP address to the TiN management CPU based on the TiN MAC address.

The Aquila controller has records of the IP addresses intended for each TiN's CPU from its own configuration. The controller continually attempts to connect to each TiN CPU via TCP session using its assigned IP address and a well known L4 port number. Once the IP address is known to a TiN's firmware, a controller message destined to that IP is trapped by an ACL rule installed by the firmware and reaches the firmware. The response Ack is sent out the same interface the packet came in from thus enabling a TCP connection between the controller and the switch CPU of the DCN-connected TiNs. The controller can then configure the DCN-connected TiN and program its routing tables.

Once the controller establishes a TCP session with a DCN-connected TiN, it uses that TiN as a *proxy TiN* (e.g., Pod 1, TiN 3) to bootstrap a directly connected *target TiN* (e.g., Pod 1, TiN 2) using the point-to-point low bandwidth Packet Management Ordered Sets (PMOS) protocol (§3.5) between TiN CPUs. A target TiN—not yet configured with its IP address—also sends DHCP discovery messages over MOS over all GNet links, which are trapped by the proxy TiN and sent over its own session to the controller. The controller in turn relays the discover message to the DHCP server, and likewise relays the DHCP response, so that a target TiN learns of its assigned IP address indirectly. The controller proceeds to configure and program the routing tables in the target TiN via the proxy TiN over PMoS. After enough routing state is programmed, the controller can establish a TCP connection to the target TiN via the GNet routing pipeline and the proxy TiN (Pod 1, TiN 3) can then be used in turn to bootstrap yet another target TiN (e.g., Pod 1, TiN 4). Once a TCP session is established

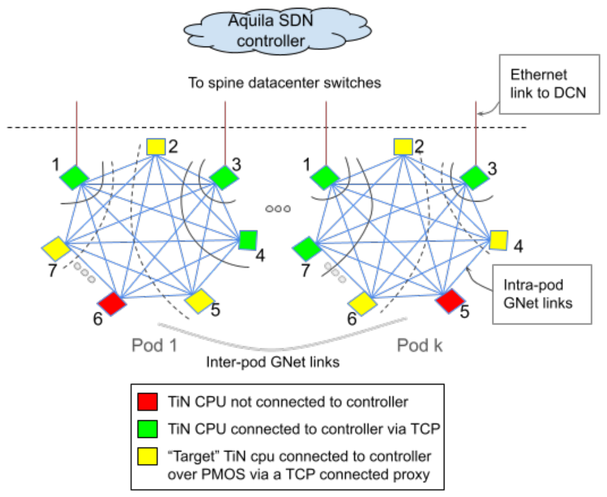
Aquila SDN controller
To spine datacenter switches
Ethernet link to DCN
Pod 1
Pod k
Inter-pod GNet links
Intra-pod GNet links
TiN CPU not connected to controller
TiN CPU connected to controller via TCP
"Target" TiN cpu connected to controller over PMOS via a TCP connected proxy

Figure 6: Inband bootstrap. The Aquila SDN controller bootstraps the TiNs inband in "waves" originating from the TiNs that are directly connected to the DCN.

with this new target TiN, it too can be used as a proxy to bootstrap a directly connected TiN (e.g., Pod 1, TiN 5), and so on.

DCN-connected TiNs typically bootstrap faster than the target TiNs, which are configured over the slower PMOS protocol leading to a distribution of bootstrap times ranging from 3 minutes to 48 minutes. Several of the waves of bootstrap occur simultaneously, resulting in a bring-up time of approximately 2.5 hours for a full-sized Clique.

5 EXPERIMENTAL RESULTS

We present a set of results examining key aspects of the Aquila network, including its data plane performance as well as its impact on application metrics.

5.1 Data Plane Performance

Aquila's data plane performance was evaluated in a prototype Aquila testbed comprised of 576 TiNs. We used 500 host machines. Two hosts share a NIC unless otherwise specified. We used two workloads, both of which run with delay-based congestion control [33]:

1. *UR*: An IP traffic generator based on a user space microkernel, *Pony Express* [37], that generates a Uniform Random traffic pattern with Poisson arrival.
2. *CliqueMap*: A key-value store [50] that uses Remote Memory Accesses (RMA) via either Pony Express or 1RMA.

For our evaluation, we used three metrics:

1. *IP Fabric RTT (μs)*: We used NIC hardware timestamps to measure Aquila fabric RTT, excluding processing and ack-coalescing delays on the remote host. This is a true measure of the transmission and queuing delay inside the Aquila fabric, both for GNet and IP components.
2. *1RMA Total Execution Latency (μs)*: the time from when the RMA command is submitted to the hardware until the hardware issues the completion for that command. This metric measures more than queuing and transmission delay in the fabric, as it includes the PCIe transaction delay on the remote side.
3. *Achieved throughput* of the network in Gbps (averaged over 30 seconds).

Latency Under Load. We examine the latency of both IP and 1RMA traffic under load. We used a CliqueMap client benchmark that issues lookups of 4 KB-sized values using RMA. By varying the QPS of the CliqueMap client on the 500 hosts, we changed the offered load per machine in a traffic pattern akin to Uniform Random. Figure 7 plots fabric RTT against offered load. It shows that the fabric latency remains under 40 μs, even when the network is close to the per machine NIC line rate of 50Gbps and it is sub-20 μs at 70% load.

1RMA is co-designed with GNet (§3.4) and Figure 8 shows that this co-design paid off with total execution time below 10 μs even under high load for 4 KB RMA reads that are generated using 500 CliqueMap clients to read from 500 CliqueMap backends.

1RMA Latency Isolation. Aquila is a unified network shared by low latency 1RMA traffic and regular IP traffic which may be latency insensitive. In our next evaluation, we show that Aquila delivers latency sensitive traffic with low tail latency despite sharing the network with IP traffic. To this end, we compare the latency of latency-sensitive traffic with and without background IP traffic in both Aquila and a conventional Ethernet network.

For the Ethernet network, we employ standard QoS techniques to isolate low-latency (or important) traffic from bulk throughput oriented traffic. We run 200 instances of CliqueMap lookups of 4 KB values at 10,000 QPS on a higher priority QoS class (H) and a UR traffic pattern with 64 KB messages with average load of 10 Gbps on a lower priority class (L). The relative egress scheduling priority between H and L classes is 8:1. The orange and cyan bars in Figure 9 show that such QoS-based schemes provide reasonable isolation for the CliqueMap traffic from the bulk IP traffic, leading to a modest increase in queuing in the fabric RTT for HiPri CliqueMap traffic, albeit with a high baseline latency.

Repeating the same experiment using 1RMA as a transport, we can see that 1RMA on Aquila offers a much lower baseline (less than 5 μs median and tail latency) despite sharing the same GNet fabric with IP traffic. High priority 1RMA traffic uses different virtual channels than low-priority Pony Express IP traffic and thus is nearly unaffected by adding the bulk traffic (blue and red bars). Even when low priority 1RMA traffic shares the virtual channels with the bulk IP traffic (yellow and green bars), the overall latency is slightly higher than 10 μs but still lower than Pony Express traffic on Ethernet networks (orange and cyan bars).

Effect of Burst Size. One of the lessons we learned in Aquila is the phenomenon of *self-congestion*. The IP network in Aquila has an injection rate of 100 Gbps per TiN while

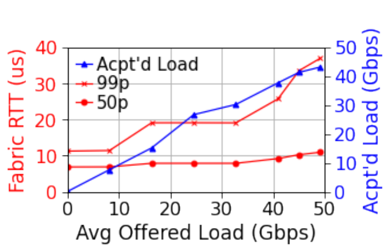

Figure 7: IP Latency vs. Load: Fabric queuing remains low under load.

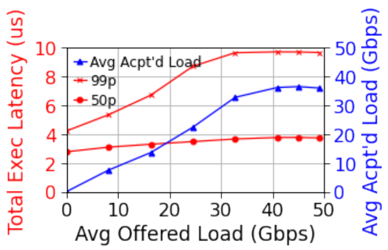

Figure 8: RMA read latency under varied 1RMA Load.

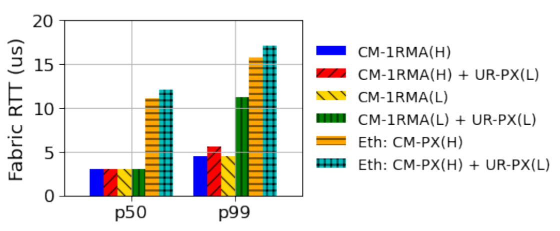

Figure 9: 1RMA Isolation: Aquila provides low latency for 1RMA traffic, even when sharing the network with IP (H = High Priority, L = Low Priority, CM = CliqueMap, PX = Pony Express)

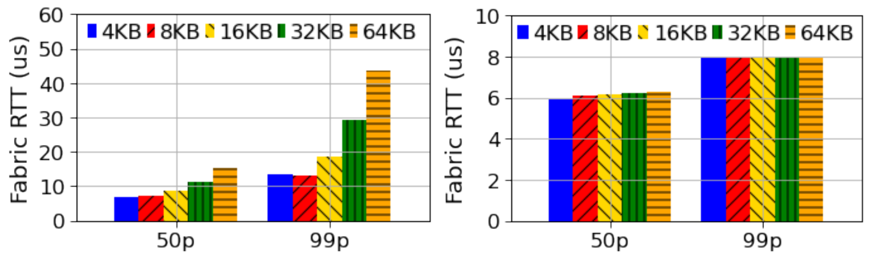

Figure 10: Effect of burstiness on queuing in a full sized Aquila (left) and a half sized Aquila (right). By keeping the injection rate constant and varying message size, we can see the effect of burstiness on queuing latency.

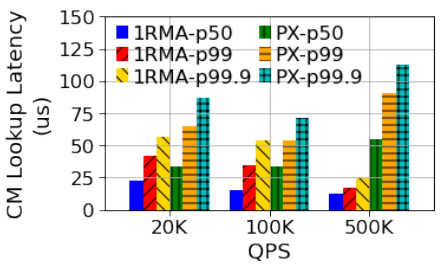

Figure 11: Effect of a cell-based RMA protocol on end-to-end CliqueMap lookup latency.

the aggregate bandwidth along the minimal paths between two pods in the full scale Aquila topology is limited to 50 Gbps. This leads to cells taking non-minimal routes even if the overall injection rate is well bellow the link rate due to bursts. We show this effect by keeping the injection rate of a point-to-point traffic at 0.6 Gbps but varying the message size of the RPC using Pony Express. Varying the message size only affects the burstiness of the injection. Figure 10 shows that as we increase message size, the tail fabric latency increases past 40 μs. However, repeating the same experiment in a half-scale version of Aquila where we have matching inter-pod bandwidth to the IP injection rate from each TiN, we see no effect of message size on queuing in the fabric. Provisioning higher minimal path bandwidth trades off better performance under bursty traffic conditions in exchange for a smaller maximum scale of the topology.

5.2 Application Impact

In order to see application impact, we compare CliqueMap lookup (of objects with 4 KB size) latency using 1RMA and Pony Express as a transport for RMAs on the Aquila network. We use O(100) backends and clients and vary queries per-second from each client. Figure 11 shows how 1RMA on Aquila cuts the median and tail latency by 50% at low QPS and by more than 300% at high QPS. As with prior work [51], higher load levels with 1RMA deliver *lower* latencies, as individual servers may dwell in low-power states at low load.

6 DISCUSSION

While our approach to Aquila's design enabled us to develop a unified low latency network fabric for datacenter networks, there were a number of challenges that we had to overcome. We highlight some of the key challenges next.

Single chip part and direct connect topology. While the single chip design delivered a sustainable development model with a modest sized team and cost efficiency for the Aquila network, the approach had a couple of key implications on the architecture and deployment. First, the single part implied that we had to deploy Aquila as a direct connect network topology because an indirect topology (such as a Clos network) was infeasible with TiN chips. While not a drawback by itself, a direct connect topology is not conducive to incremental deployment. Secondly, the evolution of the NIC and the switch architectures were coupled together from a multi-generational roadmap standpoint.

For simplicity, we designed the Aquila Clique as a homogeneous unit of deployment without an intent to mix hardware from different generations. Moreover, the networking footprint for the entire Clique (up to 24 racks housing all TiN cards as well as the networking fiber) was designed to be deployed up front and host machines could be incrementally populated on demand. With an indirect topology, a small number of network racks (e.g., 4) could be pre-deployed with server racks deployed incrementally. With a direct topology, all server racks (potentially without servers) had to be pre-deployed. Further, for a given optical technology, an indirect topology supports more deployment flexibility: all server racks need only be within a (say) 100m radius of the network racks. With our direct topology, care had to be taken to lay out the rack footprint such that the GNet fiber length between all rack pairs was within the budget of (say) 100m.

Self congestion due to thin minimal path links. The scale of a Dragonfly topology can be increased until we have only a single global link between each pair of pods. However, a mismatch between the injection bandwidth from a TiN and the pod-to-pod bandwidth leads to *self-congestion* where, even at low loads and especially for large MTU packets, some cells may be routed minimally while others may traverse non-minimal paths. As a result, there is some variance in latency introduced due to cell and packet reassembly even for point-to-point flows at low average loads.

For our initial Aquila prototype, we chose a Clique size of 576 TiNs where the pod-to-pod bandwidth was 2x25Gbps which was 1/4 the maximum injection bandwidth of 200Gbps for each TiN, a balance between Clique scale and self-congestion in the Dragonfly configuration. Further, we tuned adaptive routing to switch from minimal to non-minimal paths to reduce the variance due to self-congestion.

Overhead of cell switching and solicitation. Cell switching and solicitation are key features in Aquila for achieving predictable, low network latency. Switching GNet cells comes with an overhead of approximately 5% due to an 8 byte GNet header for each 160 byte GNet cell. The RTS/CTS solicitation for each IP packet incurs a latency overhead of an extra round trip through the network though the RTS/CTS cells get high priority through the GNet network and the overhead is further mitigated for packets with large MTU. We considered both these overheads acceptable in exchange for low tail latency even at high injected loads. Considering a larger GNet cell size as well as the ability to not incur solicitation overhead at low loads are techniques we are investigating to further mitigate these overheads.

Debugging a cell switched network. Since the Aquila Clique is not an IP routed fabric internally, standard debug tools such as `traceroute` only show 1 hop through the entire Aquila fabric. To debug data blackholes in Aquila, we implemented a *cell tracing* capability in TiN. Cells that are marked with a bit are sampled by each TiN in the cell's path and sent to a central collector over UDP. The collector can then stitch the path of the constituent cells of a packet and triangulate any mis-configured or faulty hardware.

Limited RAM on TiN and low level firmware API. To save cost and board space, we provisioned just 2MB of RAM for the firmware running on the TiN chip, which led us to a custom firmware implementation. Firmware development added significantly to the development effort, since many basic facilities had to be customized or re-implemented (e.g., logging, memory allocation, and flash storage).

The decision to expose a register level API to the SDN controller for programming the TiN chip had the benefit of shifting complexity away from the resource constrained firmware as well as simplifying the capability to upgrade firmware with Non-Stop Forwarding (NSF). It also meant that new features could be implemented without needing to roll out a new firmware version since all features of the hardware were exposed. A challenge with this approach was maintaining this interface across multiple hardware generations, since the SDN controller would need to be aware of the register level details of each chip.

For future designs, we are investigating adding more compute to the NIC so that it can be Linux based. Adding RaspberryPi equivalent compute to each NIC is likely to minimally increase the per unit cost relative to the expected gains in development velocity. Additionally, more compute will unblock the use of an API with a higher level of abstraction, such as P4 Runtime [24].

Legacy Applications Performance. While Aquila delivered significant application performance improvements (§5) for the co-designed case, such as CliqueMap with 1RMA, it did not have a significant positive impact on legacy applications. We observed that legacy application's tail latency is dominated by the host software stack, including thread wake up latency. Moreover, with IP software stacks, RTS queue length is governed by the host congestion control algorithms rather than the GNet fabric cell latency. Looking forward, we are shifting transport and network protocols to natively take advantage of future-looking hardware improvements, creating an interesting tension where the substantial software investment would likely not be a net positive until newly designed hardware is deployed across the majority of the fleet.

7 RELATED WORK

Topology and Cell-switching. Aquila uses a direct-connect topology, Dragonfly [30]. The Cray Cascade system [17] utilizes a Dragonfly topology as the basis for an HPC fabric. This design uses a high radix switch with 4 integrated host interfaces, using a proprietary packet format and virtual cut-through. The gateway to Ethernet networking requires processing nodes connected to both types of network. Aquila differs from this system (and from work on Flattened Butterflies [31] and HyperX [3]) by using the topology as a cell fabric, as opposed to a packet network with virtual cut-through. JellyFish [52] is a random-graph topology with its own challenges of deployment. Sirius [6] is a flat-topology with similar goals to Aquila but it utilizes optical circuit switching rather than cell or packet switching. Early ATM networks provided Ethernet-on-ATM [26]. More recently, Stardust [56] employed the idea of cells to give a higher effective switch radix by using single lane channels in the fabric.

Low latency networking protocols. Infiniband [10] implements an alternative networking stack to Ethernet/IP optimized for lower latency. This provides a flow controlled, lossless packet level protocol, a reliable transport implementation, and a complete set of messaging and RDMA operations. Although inter-operation with Ethernet networks for IP traffic can be implemented by gateway functions, Infiniband is commonly used as a dedicated HPC network. SRD [47] (Scalable Reliable Datagram) is an alternate transport protocol layered over IP datagrams that is used in conjunction with EFA (Elastic Fabric Adapter) by a Cloud computing provider to provide lower latency communications services for HPC applications. This has the advantage of being able to use standard Ethernet switches at some cost in minimum achievable latency. While Aquila uses cell-based adaptive routing, SRD uses source-based adaptive multi-pathing.

Congestion control. Solicitation is one of the key elements of GNet for controlling congestion in the GNet fabric. A few recent congestion control schemes such as Homa [42], NDP [25], ExpressPass [11], pHost [19] and Stardust [56] use a receiver-driven solicitation scheme, similar to that of GNet, to avoid incast congestion and achieve low latency. Aquila's solicitation controls the transfer of IP packets from buffers at the GNet fabric edge and does not directly control the IP NIC. This means it can handle both gateway and host interface IP traffic, but it requires host-based congestion control [33] to cause the traffic sources to back off in the event of congestion.

Control-plane. Aquila's control plane was designed with a distributed software defined control-plane. Most of the previous SDN controllers, Onix [32], ONOS [9], Flowlog [43], Ravel [54] assume routing of IP traffic and rely on OpenFlow to program switches. Aquila's control plane introduces a lower-level communication protocol from a Switch Frontend module to control light embedded switch controllers. The table-based design in [18, 43, 54] allowed for extending routing and sequencing to support GNet flows in addition to IP flows.

8 CONCLUSION

In this paper we present Aquila, our first foray into tightly-coupled networks (Cliques) integrated within the datacenter networking ecosystem realizing Clique-scale resource disaggregation and predictable, low-latency communication. Our primary goal is to advocate for a new design architecture for datacenter networking around Cliques and to encourage new research and development in tightly-coupled networking in support of high-performance computing, ML training, and network disaggregation while simultaneously interoperating with traditional TCP/IP/Ethernet traffic at datacenter scale. We believe our experience, both positve and negative, with the Aquila prototype will set the foundation for future exploration in this space.

Acknowledgments We would like to thank David Culler, John Wilkes, David Wetherall, the anonymous NSDI reviewers and our shepherd, Brent Stephens, for providing valuable feedback. Aquila was a multi-year effort at Google that benefited from an ecosystem of support and innovation. Many contributed to the work, including but not limited to Adam Jesionowski, Alan Lam, Alex Smirnov, Amir Salek, Brandon Ripley, Chip Killian, Daniel Nelson, David Wickeraad, Deepak Arulkannan, Deepak Lall, Duncan Tate, Jakov Seizovic, Jeffery Seibert, Jennie Hughes, Joe Love, Kamran Torabi, Luiz Mendes, Matt Maxwell, Matthew Beaumont-Gay, Philippe Selo, Phillip La, Ranjan Bonthala, Robin Zhang, Scott Berkman, Sean Clark, Shaun Tran, Simon Sabato, Steven Knight, Trevor Switkowski, Tri Nguyen, Warren James, Wilson Lee, Yousuf Haider, and Zhenchuan Pang.

REFERENCES

[1] Freertos: Real-time operating system for microcontrollers. https://www.freertos.org/. Accessed: 2022-02-28.

[2] Lwip: A lightweight tcp/ip stack. https://savannah.nongnu.org/projects/lwip/. Accessed: 2022-02-28.

[3] Jung Ho Ahn, Nathan Binkert, Al Davis, Moray McLaren, and Robert S Schreiber. Hyperx: topology, routing, and packaging of efficient large-scale networks. In *2009 SC Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11. IEEE Computer Society, 2009.

[4] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. *SIGCOMM Comput. Commun. Rev.*, 38(4):63–74, August 2008.

[5] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 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*, SIGMETRICS '12, pages 53–64, New York, NY, USA, 2012. ACM.

[6] Hitesh Ballani, Paolo Costa, Raphael Behrendt, Daniel Cletheroe, Istvan Haller, Krzysztof Jozwik, Fotini Karinou, Sophie Lange, Kai Shi, Benn Thomsen, and Hugh Williams. Sirius: A flat datacenter network with nanosecond optical switching. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 782–797, New York, NY, USA, 2020. Association for Computing Machinery.

[7] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A general approach to network configuration verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 155–168, 2017.

[8] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. Don't mind the gap: Bridging network-wide objectives and device-level configurations. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 328–341, 2016.

[9] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, et al. Onos: towards an open, distributed sdn os. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 1–6, 2014.

[10] Rajkumar Buyya, Toni Cortes, and Hai Jin. *An Introduction to the InfiniBand Architecture*, pages 616–632. 2002.

[11] Inho Cho, Keon Jang, and Dongsu Han. Credit-Scheduled Delay-Bounded Congestion Control for Datacenters. In *Proceedings of the ACM SIGCOMM 2017 Conference*, SIGCOMM '17, pages 239–252, New York, NY, USA, 2017. ACM.

[12] David D. Clark, John Wroclawski, Karen R. Sollins, and Robert Braden. Tussle in cyberspace: Defining tomorrow's internet. *IEEE/ACM Trans. Netw.*, 13(3):462–475, June 2005.

[13] Paolo Costa, Hitesh Ballani, Kaveh Razavi, and Ian Kash. R2c2: A network stack for rack-scale computers. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 551–564, 2015.

[14] William J Dally. Virtual-channel flow control. *ACM SIGARCH Computer Architecture News*, 18(2SI):60–68, 1990.

[15] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, February 2013.

[16] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, Seattle, WA, April 2014. USENIX Association.

[17] G. Faanes, A. Bataineh, D. Roweth, T. Court, E. Froese, B. Alverson, T. Johnson, J. Kopnick, M. Higgins, and J. Reinhard. Cray cascade: A scalable hpc system based on a dragonfly network. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–9, 2012.

[18] Andrew Ferguson, Steve Gribble, Chi-Yao Hong, Charles Killian, Waqar Mohsin, Henrik Muehe, Joon Ong, Leon Poutievski, Arjun Singh, Lorenzo Vicisano, Richard Alimi, Shawn Shuoshuo Chen, Mike Conley, Subhasree Mandal, Karthik Nagaraj, Kondapa Naidu Bollineni, Amr Sabaa, Shidong Zhang, Min Zhu, and Amin Vahdat. Orion: Google's software-defined networking control plane. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, 2021.

[19] Peter X. Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. pHost: Distributed Near-optimal Datacenter Transport over Commodity Network Fabric. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '15, pages 1:1–1:12, New York, NY, USA, 2015. ACM.

[20] Marina García, Enrique Vallejo, Ramón Beivide, Miguel Odriozola, and Mateo Valero. Efficient routing mechanisms for dragonfly networks. In *2013 42nd International Conference on Parallel Processing*, pages 582–592. IEEE, 2013.

[21] Ramesh Govindan, Ina Minei, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. Evolve or die: High-availability design principles drawn from googles network infrastructure. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 58–72, New York, NY, USA, 2016. Association for Computing Machinery.

[22] R. L. Graham, D. Bureddy, P. Lui, H. Rosenstock, G. Shainer, G. Bloch, D. Goldenerg, M. Dubman, S. Kotchubievsky, V. Koushnir, L. Levi, A. Margolin, T. Ronen, A. Shpiner, O. Wertheim, and E. Zahavi. Scalable hierarchical aggregation protocol (sharp): A hardware architecture for efficient data reduction. In *2016 First International Workshop on Communication Optimizations in HPC (COMHPC)*, pages 1–10, 2016.

[23] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. Vl2: A scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, SIGCOMM '09, page 51–62, New York, NY, USA, 2009. Association for Computing Machinery.

[24] The P4.org API Working Group. P4 Runtime Specification. https://p4.org/p4runtime/spec/v1.2.0/P4Runtime-Spec.html, 2020.

[25] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichik, and Marcin Mojcik. Re-architecting Datacenter Networks and Stacks for Low Latency and High Performance. In *Proceedings of the ACM SIGCOMM 2017 Conference*, SIGCOMM '17, pages 29–42, New York, NY, USA, 2017. ACM.

[26] Hong Linh Truong, W. W. Ellington, J. Y. Le Boudec, A. X. Meier, and J. W. Pace. Lan emulation on an atm network. *IEEE Communications Magazine*, 33(5):70–85, 1995.

[27] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a globally-deployed software defined wan. In

Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM, SIGCOMM '13, page 3–14, New York, NY, USA, 2013. Association for Computing Machinery.

[28] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, page 1–12, New York, NY, USA, 2017. Association for Computing Machinery.

[29] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 113–126, San Jose, CA, April 2012. USENIX Association.

[30] John Kim, Wiliam J Dally, Steve Scott, and Dennis Abts. Technology-driven, highly-scalable dragonfly topology. In *2008 International Symposium on Computer Architecture*, pages 77–88. IEEE, 2008.

[31] John Kim, William J Dally, and Dennis Abts. Flattened butterfly: a cost-efficient topology for high-radix networks. In *Proceedings of the 34th annual international symposium on Computer architecture*, pages 126–137, 2007.

[32] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, et al. Onix: A distributed control platform for large-scale production networks. In *OSDI*, volume 10, pages 1–6, 2010.

[33] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan M. G. Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, David Wetherall, and Amin Vahdat. Swift: Delay is simple and effective for congestion control in the datacenter. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 514–528, New York, NY, USA, 2020. Association for Computing Machinery.

[34] Sergey Legtchenko, Nicholas Chen, Daniel Cletheroe, Antony Rowstron, Hugh Williams, and Xiaohan Zhao. Xfabric: A reconfigurable in-rack network for rack-scale computers. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 15–29, 2016.

[35] Yuliang Li, Gautam Kumar, Hema Hariharan, Hassan Wassel, Peter Hochschild, Dave Platt, Simon Sabato, Minlan Yu, Nandita Dukkipati, Prashant Chandra, and Amin Vahdat. Sundial: Fault-tolerant clock synchronization for datacenters. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1171–1186, 2020.

[36] Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, and Thomas Anderson. F10: A fault-tolerant engineered network. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 399–412, Lombard, IL, April 2013. USENIX Association.

[37] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, and et al. Snap: A microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 399–413, New York, NY, USA, 2019. Association for Computing Machinery.

[38] Nick McKeown. The islip scheduling algorithm for input-queued switches. *IEEE/ACM transactions on networking*, 7(2):188–201, 1999.

[39] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling Innovation in Campus Networks. 38:69–74, 2008.

[40] Michael Mitzenmacher, Andréa W. Richa, and Ramesh Sitaraman. The power of two random choices: A survey of techniques and results. In *Handbook of Randomized Computing*, pages 255–312. Kluwer, 2000.

[41] Jeffrey C Mogul, Drago Goricanec, Martin Pool, Anees Shaikh, Douglas Turk, Bikash Koley, and Xiaoxue Zhao. Experiences with modeling network topologies at multiple levels of abstraction. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 403–418, 2020.

[42] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 221–235, New York, NY, USA, 2018. ACM.

[43] Tim Nelson, Andrew D Ferguson, Michael JG Scheer, and Shriram Krishnamurthi. Tierless programming and reasoning for software-defined networks. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 519–531, 2014.

[44] Open Networking Foundation. Mission of open networking foundation. https://opennetworking.org/mission/, 2021. Accessed: 2021-03-08.

[45] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the Social Network's (Datacenter) Network. In *Proceedings of the ACM SIGCOMM 2015 Conference*, SIGCOMM '15, pages 123–137, New York, NY, USA, 2015. ACM.

[46] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan RK Ports, and Peter Richtárik. Scaling distributed machine learning with in-network aggregation. April 2021.

[47] Leah Shalev, Hani Ayoub, Nafea Bshara, and Erez Sabbag. A cloud-optimized transport protocol for elastic and scalable hpc. *IEEE Micro*, 40(6):67–73, 2020.

[48] Arjun Singh. *Load-balanced routing in interconnection networks*. PhD thesis, Stanford University, 2005.

[49] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Hanying Liu, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *SIGCOMM '15*, 2015.

[50] Arjun Singhvi, Aditya Akella, Maggie Anderson, Rob Cauble, Harshad Deshmukh, Dan Gibson, Milo MK Martin, Amanda Strominger, Thomas F Wenisch, and Amin Vahdat. Cliquemap: productionizing an rma-based distributed caching system. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 93–105, 2021.

[51] Arjun Singhvi, Aditya Akella, Dan Gibson, Thomas F Wenisch, Monica Wong-Chan, Sean Clark, Milo MK Martin, Moray McLaren, Prashant Chandra, Rob Cauble, et al. 1rma: Re-envisioning remote memory access for multi-tenant datacenters. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 708–721, 2020.

[52] Ankit Singla, Chi-Yao Hong, Lucian Popa, and P Brighten Godfrey. Jellyfish: Networking data centers randomly. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 225–238, 2012.

[53] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015.

[54] Anduo Wang, Xueyuan Mei, Jason Croft, Matthew Caesar, and Brighten Godfrey. Ravel: A database-defined network. In *Proceedings of the Symposium on SDN Research*, pages 1–7, 2016.

[55] Junlan Zhou, Malveeka Tewari, Min Zhu, Abdul Kabbani, Leon Poutievski, Arjun Singh, and Amin Vahdat. WCMP: Weighted Cost Multipathing for Improved Fairness in Data Centers. In *Proceedings of the Ninth European Conference on Computer Systems*, page Article No. 5, 2014.

[56] Noa Zilberman, Gabi Bracha, and Golan Schzukin. Stardust: Divide and conquer in the data center network. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 141–160, 2019.

A HARDWARE PACKAGING DETAILS

Incremental deployment is a much more significant consideration for datacenter systems than for supercomputers which are typically installed as a single system, or in a number of predefined phases. Incremental network deployment is challenging for the Dragonfly topology, where growing the size of the fabric requires the topology to be reconfigured to fully exploit the available chip bandwidth. To avoid recabling for expansion, which is hard to reconcile with the availability requirements of a datacenter, we developed a packaging strategy that allows all the networking infrastructure to be landed as one initial deployment, with the servers being populated incrementally as required.

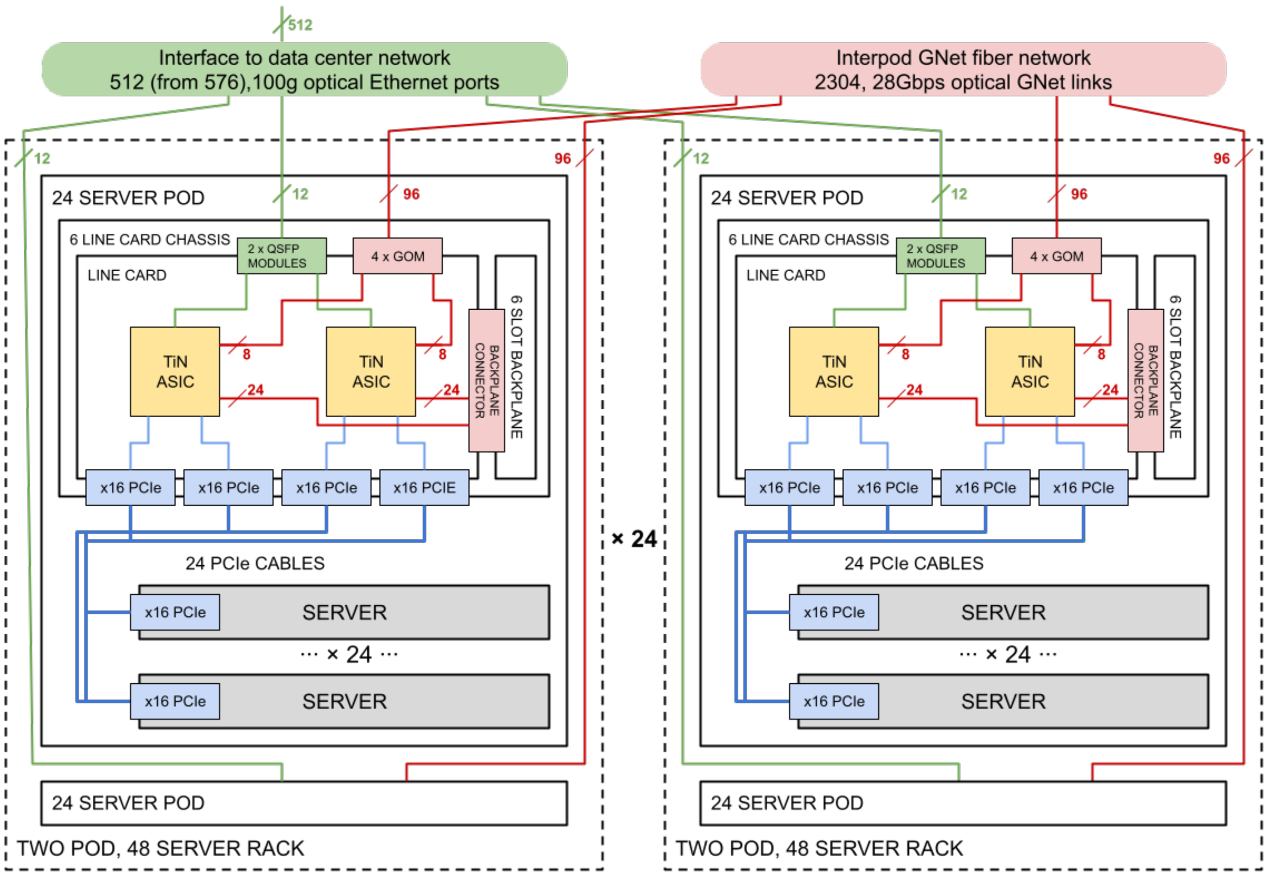

Figure 12: Aquila Clique with 1152 servers, in 24 racks.

A second key consideration was whether to use a blade based system, with a combined server and networking packaging solution, or to work with our existing servers designed around conventional NICs. The latter approach was chosen to avoid having to support two packaging variants of each different server type. These two decisions broadly determined our packaging design.

The physical design (Figure 12) supports up to 48 machines per rack, organized as two pods of 24 servers. The Aquila networking for each pod is provided by a switch chassis containing 12 TiN ASICs, on 6 line cards. The first level interconnect of the Dragonfly is implemented in copper on the switch chassis backplane. Servers are connected to the switch chassis using a cabled x16 Gen 3 PCIe bus. Sideband signals on the cable carry the independent machine management interface from the TiN chip that connects to the server's NC-SI port.

The overall Aquila Clique consists of 24 racks. The connectivity between the racks is optical using custom low cost VCSEL based 4 channel GNet optical modules, 4 per line card. This gives a total of 96 optical GNet connections for the global interconnect level of the Dragonfly from each pod. As there are a total of 48 pods in a clique there are two optical GNet global links between any pod pair. If we connected these directly with two channel fiber ribbons this would require 47x48/2 = 1128 unique interpod cables to be connected. To simplify the rack to rack cabling we use fiber shuffles within groups of 4 pods to consolidate into wider fiber ribbons allowing the use of 8 GNet link, MPO16 fiber cables. This reduces the rack to rack cabling to 66 4-cable bundles running between 12 pairs of racks greatly simplifying the fiber deployment.

The total number of available 100g Ethernet ports available for connection to the data center spine network from the TiN ASICs is 576. 24 of these are used for rack management. Either 256 or 512 ports are connected to the higher level Ethernet fabric with the remaining 40 ports unused.

A.1 Failure Domains

A key consideration of the Aquila architecture was to reduce the blast radius of any networking component failure. In a conventional network the loss of a TOR impacts all the attached servers; this could be as many as 48 machines for a high radix switch device. In contrast, with the Aquila architecture, loss of a TiN ASIC impacts a maximum of two servers. In practice because the physical packaging solution uses a pair of TiN ASICs on a single line card, the effective blast radius for a repair operation can be up to four servers if 2 servers share a TiN. A switch chassis failure impacts a maximum of 24 servers, however the only chassis components with a significant failure rate are the fans, and N+2 fan redundancy

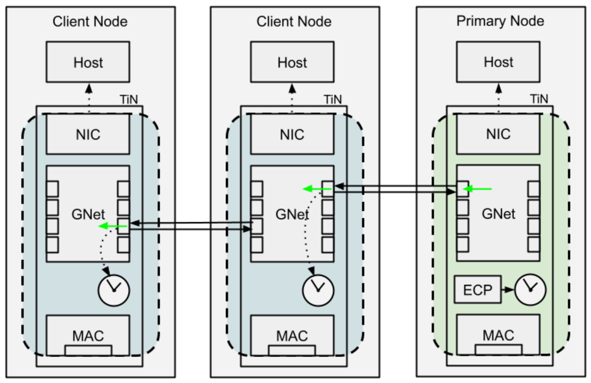

Figure 13: Aquila Clock Sync.

is implemented to minimize the possibility of a chassis level failure.

B CLOCK SYNCHRONIZATION

B.1 Overview

The timesync protocol on Aquila was designed with the aim of keeping the software overhead for timesync low while also providing a tight bound on the notion of current time across the TiNs in the clique. This protocol maintains a single primary clock in the clique against which clients are synchronized purely in hardware (Figure 13). Synchronization is carried out over the GNet links on the switch side of the TiN by transmitting information as lightweight, 8-Byte “ordered sets” between cells, a class of which (TimeSync) are defined for Clique time synchronization. Clients in the host, expecting an IEEE 1588-like protocol to maintain time in the NIC, are able to query the value of this clock. The hardware also corrects for link delays between neighboring TiNs and for time spent within the chip while waiting for a gap between cells to get on the link.

B.2 Implementation

The Timesync hardware on TiN maintains the current time by counting cycles of the core clock along with status bits which tracks several parameters that indicate the accuracy of the clock. The value of time is also updated by the reception of timesync messages from the neighboring TiN if the current TiN has been configured to be a client node in the time distribution network. The protocol relies on software to set up this time distribution tree [35].

Once the time distribution tree is configured, the TiN transmits TimeSync ordered sets on a configured number of output GNet links at a fixed interval (typically, about 100us). On a client node, an incoming TimeSync message also causes an update to be sent downstream even if the configured interval between messages has not expired. This is to ensure that even the farthest nodes in the time distribution tree do not drift much from the primary node.

The TimeSync message cannot interrupt a cell on the wire, so the ordered set can wait up to 128ns to get onto the wire. Regardless of the delay, the ordered set indicates the actual time of transmission (+/- 2.5ns) by incrementing the value of current time in the TimeSync message for each cycle that it waits to get onto the wire, including flight time across the chip from the hardware clock, arbitration time to get onto the wire, etc. Each receiving TiN is configured to receive TimeSync messages only on a single port and it adjusts for any on-chip delays to get the TimeSync message to the hardware clock along with the delay through the GNet channel.

The delay through the GNet channel is configured on the receiver by running round trip delay measurement at the time of setting up of the time distribution tree. This is done by the GNet ports by putting them in a “latency measurement” mode where the neighbors exchange special ordered sets and reflect the delay through the channel to software as a status.

On reception of a Timesync message, the client node, checks the validity of the message through comparison of status bits transmitted with the message and the difference between the incoming time and the current time against a configurable threshold. The update to current time is only applied when valid Timesync messages are received and if enough invalid messages are seen, the client node signals that a failure is detected. The protocol relies on software to take action once failure is detected