STREGA: An HTTP Server for FPGAs

FABIO MASCHI and GUSTAVO ALONSO, Systems Group, Department of Computer Science, ETH Zurich, Switzerland

The computer architecture landscape is being reshaped by the new opportunities, challenges, and constraints brought by the cloud. On the one hand, high-level applications profit from specialised hardware to boost their performance and reduce deployment costs. On the other hand, cloud providers maximise the CPU time allocated to client applications by offloading infrastructure tasks to hardware accelerators. While it is well understood how to do this for, e.g., network function virtualisation and protocols such as TCP/IP, support for higher networking layers is still largely missing, limiting the potential of accelerators. In this article, we present STREGA, an open source[1] light-weight Hypertext Transfer Protocol (HTTP) server that enables crucial functionality such as FPGA-accelerated functions being called through a RESTful protocol (FPGA-as-a-Function). Our experimental analysis shows that a single STREGA node sustains a throughput of 1.7 M HTTP requests per second with an end-to-end latency as low as 16 µs, outperforming nginx running on 32 vCPUs in both metrics, and can even be an alternative to the traditional OpenCL flow over the PCIe bus. Through this work, we pave the way for running microservices directly on FPGAs, bypassing CPU overhead and realising the full potential of FPGA acceleration in distributed cloud applications.

CCS Concepts: • **Hardware → Networking hardware**; **Hardware accelerators**; • **Computer systems organization → Client-server architectures**; *Cloud computing;*

Additional Key Words and Phrases: Network on chip, FPGA, distributed systems, disaggregated accelerator, HTTP, Webserver, RESTful API

ACM Reference format:
Fabio Maschi and Gustavo Alonso. 2024. STREGA: An HTTP Server for FPGAs. *ACM Trans. Reconfig. Technol. Syst.* 17, 1, Article 3 (January 2024), 27 pages.
https://doi.org/10.1145/3611312

1 INTRODUCTION

The cloud has evolved into a highly distributed architecture pooling a potentially large number of commodity machines to complete a given task. As a result, the past few decades have seen a plethora of distributed software platforms being proposed to facilitate the development of such applications, notably Spark [69], Flink [8], Apache Drill [24], Kafka [33], and Memcached [19], to name but a few. This trend toward increasingly decentralised architectures has now reached

[1] Available together with the additional tools at https://github.com/fpgasystems/strega

This research was supported in part by AMD under the Heterogeneous Accelerated Compute Clusters (HACC) program.
Authors' address: F. Maschi and G. Alonso, Systems Group, Department of Computer Science, ETH Zurich, Stampfenbachstrasse 114, Zurich, Switzerland, 8092; e-mails: {fabio.maschi, alonso}@inf.ethz.ch.

1936-7406/2024/01-ART3 $15.00
https://doi.org/10.1145/3611312

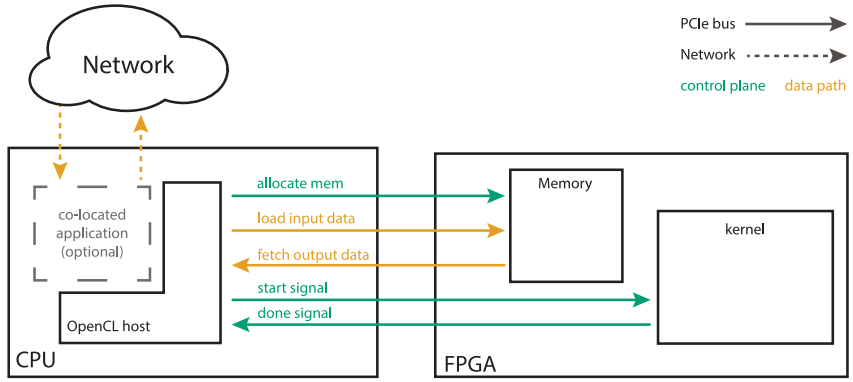

Fig. 1. Traditional flow for FPGA acceleration in distributed systems. Client requests come from the network, hop into the CPU, and are routed to the FPGA kernel. Both control plane and data path navigate through the PCIe bus. Additional applications might be co-located in the CPU to either fully utilise the resource, or to feed more workload into the FPGA. The OpenCL host must be maintained to remain compatible with the whole stack: OS, CPU architecture, FPGA vendor, board model, shell, and driver versions.

an extreme in the form of *microservices* [21, 51, 62]: comparatively small, independent software modules that communicate via light-weight APIs and are dynamically combined to implement complex applications. Microservices offer numerous advantages, such as independent deployment and invocation, increased flexibility, simplified software lifecycle, and enhanced scalability and resilience. Major cloud providers, including Google [23], Microsoft [2], and AWS [58], provide extensive support for microservices.

At the same time, the economies of scale found in the cloud make the utilisation of heterogeneous hardware not only economically feasible but also desirable from an efficiency standpoint. Thus, the cloud today comprises many different forms of computing nodes, disaggregated layers, and processing opportunities, including GPU clusters, TPUs, and SmartNICs that go well beyond traditional CPUs. FPGAs are an important part of this evolution, as they are increasingly used not only for application acceleration but also to offload infrastructure tasks away from the CPU.

Successful as FPGAs have been so far, we are still far from exploiting their full potential due to limitations in their usage and deployment as a result of the lack of support for interfaces that are compatible to those predominantly used in data centres. Traditionally, an FPGA-accelerated function is separated into two components: the host application running on the CPU and the kernel executed on the FPGA. Both communicate trough the PCIe bus, much like GPU kernels, as illustrated in Figure 1: (i) memory is first allocated on the device, (ii) data are transferred from the host to the device, (iii) the kernel is executed, and (iv) the CPU fetches the result data from the device memory. In the context of distributed systems deployed in the cloud, the consequence of this flow is that client requests must navigate *through* the CPU to be accelerated by the FPGA, imposing not only considerable communication overhead [41] but more importantly a very tight coupling between both the host and the accelerator [39].

The trend is nowadays to move load away from the CPU, allowing it to focus on running conventional applications [18, 50]. If part of the CPU needs to be used every time an accelerator is used, then the appeal of the accelerator diminishes. This is one reason why for, e.g., GPUs and TPUs, the dominant design is to provide several of them together under the control of a single CPU so as to avoid allocating CPU cycles for each one of the accelerators. Although this is not strictly necessary for FPGAs, as they often have a network interface and are able to directly interact with applications, the lack of interfaces and levels of abstraction compatible with cloud software and specifications

limits FPGAs from being utilised as first-class processing units. In part this is why cloud offerings have been restricted to a limited number of FPGA-CPU configurations. Amazon Web Services offers FPGA boards connected to servers with specific numbers of vCPUs [57], but there is no option to connect a powerful CPU to an FPGA device. This limitation makes the use of FPGAs too expensive when a single FPGA can consume the workload produced by several CPUs [39].

Rather than be subordinated to a CPU, accelerators have shown to be very effective when placed directly on the network, such as a SmartNIC, providing services and functions without CPU intervention. Microsoft AccelNet [18] is a great example of a large-scale FPGA-based SmartNIC deployment that leverages the high bandwidth capacity and low latency of FPGAs to implement network infrastructure tasks.

This article explores the direct accessibility and compatibility of network-attached FPGAs with existing cloud software, particularly as a first step toward running microservices on FPGAs. Doing so enables FPGA-supported applications to be directly available, bypassing the high overhead of going through a CPU. In microservice architectures involving potentially thousands of independent microservices, this overhead is not only a matter of latency but also of wasted resources, as the CPU is devoted to relaying data to the accelerator rather than performing useful work. The key to accomplish this is to make the FPGA appear like any other element of a microservice architecture, which requires expanding its networking capabilities. While there are a number of studies in the literature on porting the network, transport, session, and application layers to hardware [5, 25, 27, 60, 61], comparatively little attention has been given to the presentation layer, which connects the network and the application. This layer enables applications to operate at a higher level of abstraction without directly handling network packages. Thus, a crucial missing piece in FPGA infrastructure is support for the **Hypertext Transfer Protocol (HTTP)** interface, which has become the *de-facto* standard for communication in the cloud and data centres.

HTTP is today far more than just the transport protocol for web browsers. It is widely used as the main RESTful interface in the cloud for a wide range of architectures and applications thanks to its cross-platform portability. Microservices [1] are commonly invoked via HTTP, as are object stores (e.g., AWS S3), serverless functions, and numerous cloud services. In fact, the most prevalent means of connecting applications in the cloud is gRPC,[2] a modern version of the traditional **Remote Procedure Call (RPC)** built on top of HTTP leveraging the benefits provided by RESTful interfaces.

Accordingly, in what follows we present Strega, an open source light-weight HTTP server for FPGAs, which enhances the communication abstraction between *decoupled* clients and FPGA-based kernels. Applications leveraging Strega can benefit from network-attached FPGA deployments, operating without allocating a CPU instance, providing a vendor- and platform-agnostic interface that enhances modularity and drastically reduces maintenance costs, achieving predictable and much faster communication latency, and processing much more requests per second than traditional invocation techniques. In such configuration (Figure 2), the function invocation becomes RESTful request-response messages, and the host application becomes a *distributed client* that can, in turn, be any other processing node of the system, be a CPU, an API Gateway, or another FPGA.

We identify four contributions in different but complementary parts of this work: (i) We present the first complete open source, HLS implementation of an application-agnostic HTTP server for FPGAs that yields orders of magnitude higher throughput and more deterministic latency than commercial CPU-based solutions; (ii) the architecture enables FPGA-CPU disaggregation in cloud deployments by providing a RESTful interface for applications accelerated on FPGAs, thereby

[2] https://grpc.io

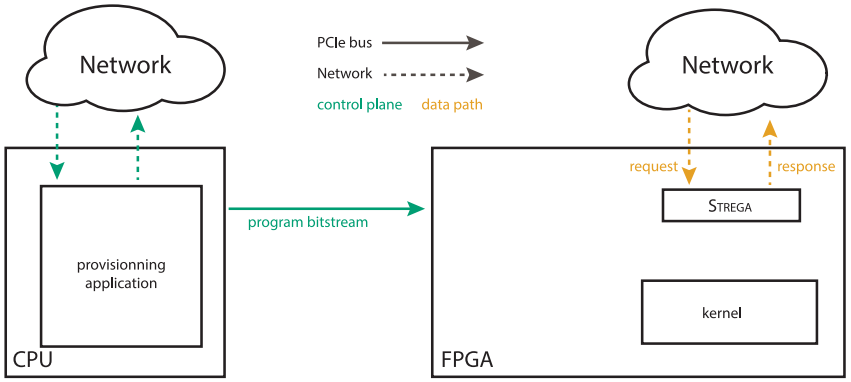

Fig. 2. Proposed design, where control plane and data path are separated: client requests come from the network using HTTP messages, being processed and answered directly in the FPGA. The host CPU becomes a simple provisioning manager, responsible for programming the FPGA with the bitstream.

increasing the cross-platform portability of the overall system; (iii) borrowing software engineering best practices, we provide an extensive testing, debugging, CI/CD, and simulation environment with automation scripts, unit tests, and mock modules that efficiently and systematically validate different design levels, facilitating the development of applications on top of STREGA; and (iv) we elaborate our findings and discuss the potential impacts of this work for both cloud providers and developers, notably in the context of serverless computing.

2 BACKGROUND AND RELATED WORK

This section provides a brief overview of the main aspects of HTTP and RESTful interfaces, as well as of related work in FPGAs for microservices and FPGA SmartNICs.

2.1 HTTP

The HTTP [16] is the *de-facto* OSI model presentation layer [26] of most online applications, covering from simple web servers serving static content, to RPC of distributed systems, and including RESTful microservices. Its dominance comes from its simplicity (stateless communication, human-friendly encoding), generality, and abstraction; which allows servers, clients, and middlewares to communicate regardless of their technology stacks.

Differently from the TCP/IP layer and other communication protocols that deal with binary information, HTTP encodes information in ASCII characters. Communication is established following the *request-response* pattern, meaning a user agent (here simply called *client*) sends a request to a server and the server returns a response. Both messages are composed of headline, headers and an optional body.

A request's headline identifies: (i) the method (such as `GET`, `POST`, `PUT`, or `DELETE`), (ii) the endpoint (e.g., `/compression`), and (iii) the protocol version (`HTTP/1.1`). Response's headline identifies (i) the protocol version and (ii) the response status (such as `403 Forbidden` or `200 OK`). Headers consist of metadata key–value pairs encoded in ASCII to enhance expressiveness for both requests and responses and can be set by the client, the server, or the application. Typically, the main payload of the transaction is found in the body, which can be encoded as text or in binary, and can be compressed by the server to save network bandwidth. From the HTTP layer perspective, each request-response is processed in a stateless manner, which favours high throughput. Thanks to the TCP/IP layer, the underlying transmission of packets are considered to be delivered in-order and reliably. If a failure or timeout occurs, then the request may be re-issued from the client side.

```
1 POST /v1/accounts HTTP/1.1
2 Content-Type: application/json;
3 Content-Length: 14
4 Connection: keep-alive
5
6 {"foo": "bar"}
```

Listing 1. HTTP request example.

Listing 1 presents a very basic example of a `POST` request to the endpoint `/v1/accounts`, where the client sends a `json` payload of 14 B. Listing 2 illustrates a possible response to the former request, where the server sends a `201 Created` status alongside with further information in the body. Applications can customise and define (i) their own set of headers, (ii) the behavioural logic based on the endpoints (for instance, having two functions `/encrypt` and `/decrypt` co-existing in the same server), and (iii) the body payload syntax. The standard defines the *structure* of the communication messages. For instance, imagine a machine learning inference application running on the FPGA exposing HTTP endpoints. A client might issue a `POST` request providing the raw binary data of a picture to be inferred. Upon reception, an FPGA kernel can process the image and respond with the list of labels identified within that image using a `json` body.

```
1 HTTP/1.1 201 Created
2 Content-Type: application/json; charset=utf-8
3 Content-Length: 38
4
5 {"location":"https://api.ethz.ch/v1/accounts/64936"}
```

Listing 2. HTTP response example.

The current version of Strega implements HTTP version 1.1 [17], an improvement over version 1.0 by keeping TCP/IP connections alive across successive requests, i.e., it has support for the `Connection: Keep-alive` header by default. The version is also the first step to implement HTTP/2.0 [4] in the future, which introduces the notion of HTTP frames in addition to the features of HTTP/1.1 implemented so far.

2.2 RESTful Interface

If HTTP can be seen as the *syntax* and behavioural pattern of how a given client and a given server communicate, then it does not establish any convention in terms of the *semantics* of each endpoint of a given application. For instance, it is possible to implement a complete API by using only `POST` requests and still be compliant to the HTTP specifications. However, how applications exchange information and the assumptions made in doing so have a large impact on both performance and how easy they are to maintain. Among the many proposal that have been made over the years on how to address this issue, the one most in use these days is **Representational State Transfer (REST)** [12, 15, 47]. REST is a set of design rules and guidelines, such as statelessness, separation of concerns, and uniform interfaces, that is usually used for the creation of scalable, maintainable, and distributed web services. The main idea behind RESTful interfaces is to allow clients to manipulate resources on the server through a well-defined set of operations, such as create, retrieve, update, and delete, which are performed using standard HTTP methods, such as `POST`, `GET`, `PUT`, and `DELETE`. In a nutshell, REST guides applications and clients to establish a common semantic for their interfaces and uses HTTP as the syntax and grammar of it. The advantages of using REST principles in microservices is that they make it easier to dynamically combine independently developed services, providing an easier basis for implementing recovery, migration, scalability through replication, and so on, independently of the actual implementation or nature of the service [28, 71].

2.3 Microservices on FPGAs

The current PCIe model of utilising FPGAs not only for kernel invocation (data path) but also for provisioning, bitstream programming, and synchronisation (control plane) has been questioned in numerous works in the literature [3, 20, 35, 42, 44, 48, 64]. By granting more abstraction in each one of those fronts, related work moved toward a direction where the common goal can be seen as offering an FPGA-as-a-Service architecture.

Lallet et al. [34] used RIFFA [30] to build a framework connecting Docker containers to FPGA kernels. They managed to mimic a microservice architecture by implementing a partial reconfiguration manager attached to the network, programming the FPGA on-demand in a multi-tenant deployment. While they moved the control path to the network, they maintained the data path of applications through the PCIe bus in an attempt to reduce latency and sustain higher throughput. Our findings show that moving the data path to the network actually reduces latency compared to PCIe end-to-end communication. Similarly, Ojika et al. [46] provide a provisioning manager for a cluster of PCIe-attached FPGA boards, where the host machine accumulates the function of global hypervisor running docker containers. Applications can then allocate FPGA kernels using their proposed custom scheduler, similar to what a load balancer would perform in a distributed system. Bacis et al. [3] created a robust hardware abstraction platform to provision, deploy and execute FPGA applications on-demand, exposing the control interfaces running on a CPU as a service. They integrate their framework with cloud solutions like Kubernetes[3] for handling application lifecycles and Prometheus[4] for profiling and monitoring. Despite the great physical layer abstraction they achieve, by using OpenCL-based methods for communicating with the FPGA, all interactions must go through the CPU.

To the best of our knowledge, there has not been related work able to show efficiency and feasibility of FPGA deployments without constraining programming methods to a very particular solution. Additionally, our work differs from the previous ones by decoupling FPGA and CPU, which naturally increases the flexibility of scalability for cloud deployments.

A different approach is that of Lazarev et al. [35], who propose an FPGA-based accelerator to offload RPC invocation and network handling from the host CPU. They take advantage of memory interconnects as an alternative to the PCIe bus, which yields lower latency for RPC payloads of size up to a cache line. Authors also present a detailed characterisation of the communication pattern of different microservice workloads. While their proposal benefits applications running on the CPU via accelerating the network stack for RPC, our solution aims at giving equivalent benefits for FPGA-based kernels. Also, imposing cache line size messages is quite a limitation in practice that we avoid with our solution.

2.4 SmartNIC FPGAs

In recent years, there have been increasing efforts to make FPGAs available over the network. Eskandari et al. [13] present a framework that offers network interfaces and communication protocol out of the box. They target the network layer of the OSI model, thus a lower one compared to the presentation layer of HTTP. They implement their communication framework in a heterogeneous system, having CPU and FPGA as processing nodes, and provide a very detailed micro-benchmark of different node communications.

In the cloud infrastructure field, Tarafdar et al. [63] mention the need of more integration efforts in the communication of heterogeneous clusters. While their main focus concerns provisioning

[3] https://kubernetes.io
[4] https://prometheus.io

mechanisms, their open stack proposal deploys network interfaces on the CPU to address PCIe-attached FPGAs resources.

Liu et al. [36] present an architecture that places the FPGA behind a CPU-based web server. This architecture yields the most flexible solution in terms of HTTP features, at the cost of allocating a CPU for the task and thus keeping a tight dependency between the CPU and the accelerator. It also suffers from the communication and synchronisation overheads imposed by the PCIe bus between the CPU and the FPGA. We implemented this approach as one of the baselines for performance comparisons of Strega in Section 4.

Brzoza-Woch and Nawrocki [6] discuss the challenges they found throughout their journey on building an FPGA web service prototype. They identify, among others, two main obstacles for the success of web-available FPGAs: (i) the hardware development and integration tools are still nowadays very bothersome to develop hardware, notably when integrating higher-level protocols, and (ii) implementation and maintenance of such solutions are too complex. While the scope of this work does not address the first issue they raised, our solution facilitates *verifying* that the tools built the expected hardware by providing c-simulation, mock engines, and hardware emulation out of the box. We also address their second concern by implementing Strega using HLS in a highly modular architecture that can be easily expanded to support new features. These two contributions provide development and testing environments that, while not comparable to those available in the software world, represent a solid starting point (see Section 3.4).

Given the number of features and high-level data processing required by the network presentation layer, first FPGA-based implementations used soft- and hard-cores (e.g., NIOS and MicroBlaze) [7, 31, 38, 54, 59, 68], but performance, resource utilisation, generality, and ease of usage were not the focus. When a soft- or a hard-core is employed for such task, the implementation burden is shifted to the hardware kernel, which has vendor-specific mechanisms and interfaces to consume HTTP requests from the soft-core [6]. Our solution exposes generic AXI4-Streams to the application together with simple and well-defined data structures that can be used in both RTL or HLS.

Field [14] has published an initial work of an intended FPGA web server in 2016. The implementation was done in VHDL and covers the TCP/IP layer as well as the HTTP. The project has never been finished and has since then been abandoned, as far as it can be ascertained from the repository.

To the best of our knowledge, there are two FPGA implementations of an HTTP server using RTL: Chang et al. [9] and Polig et al. [49], from IBM Research. While the former is very primitive and does not support realistic constraints (i.e., a single TCP/IP connection can be established at a time), IBM Research has several contributions in the field of FPGAs-as-a-Service in the cloud using their RESTful interface for the cloudFPGA platform [52, 65, 66]. Their approach relies on a static OpenAPI specification to generate the API endpoints to be used by the application. This means that the generated bitstream is fixed to a particular set of endpoints, and any changes would require a new bitstream to be generated. With such a constraint and no further details, it is uncertain if their approach supports dynamic value endpoints, such as `id` in `GET /account/:id`. In contrast, our implementation maintains the generality of the HTTP server semantics and pushes the business logic of validating endpoints to the application if needed. Our approach also differs from Reference [49] by supporting HTTP headers for both requests and responses, exposing request headers to the application, and allowing it to define custom key–value pairs for composing HTTP response headers. Finally, Strega has been conceived targeting boards intended for data centre applications, such as the one from the AMD's Alveo series.[5]

[5] https://www.xilinx.com/content/xilinx/en/products/boards-and-kits/alveo.html

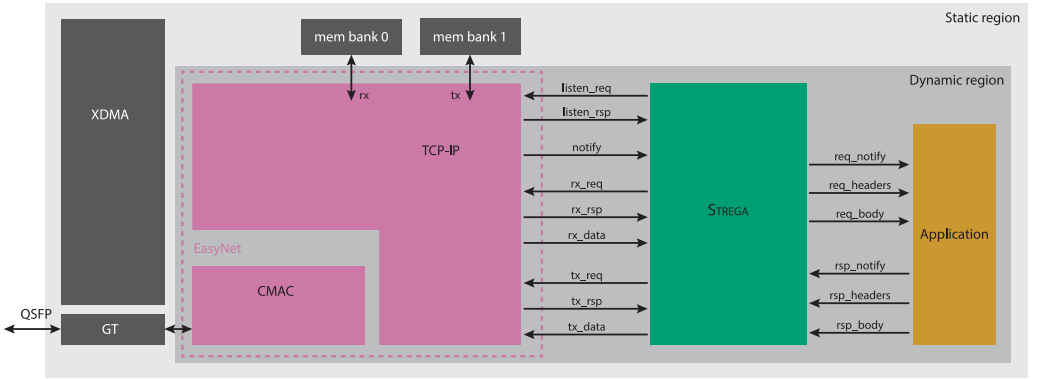

Fig. 3. Block diagram of the overall system architecture in the FPGA. In dark gray the static elements and I/O pins of the FPGA, notably network adaptors and memory banks. In pink, TCP-IP and CMAC kernels from EasyNet [25]. STREGA and the applications compose two independent kernels. Arrows to and from STREGA consist of AXI4-Stream uni-directional channels.

3 DESIGN

In this section, we detail the design of STREGA in the context of the FPGA shell and external kernels, as well as its internal architecture. We also discuss the support provided to developers building applications on top of STREGA, an important topic somewhat underrepresented in the reconfigurable computing field.

3.1 High-level Architecture

STREGA has been built on top of EasyNet [25], an HLS 100 Gbit/s networking stack developed from open source TCP/IP stacks for FPGAs [53, 60], that runs on the Vitis Development Platform.[6] STREGA runs as an independent kernel, sitting between the TCP/IP and Application kernels (Figure 3). This organisation allows better modularity and thus ease of maintenance, separating the concerns of each layer.

STREGA I/O interfaces consist of nine AXI4-stream uni-directional channels connecting it to the interface provided by the TCP/IP kernel [25]. Through them, STREGA can open and listen to ports (0-32767), receive notifications, and read and write TCP/IP messages. STREGA is connected to the application kernel through six AXI4-Stream channels, three for the request flow and three for the response flow. By exposing to the application simple, yet well-defined streaming interfaces, along with HLS data structures, it becomes relatively simple for applications to use STREGA.

Given the nature of an HTTP server, the kernel is implemented as a service, i.e., a *free-running* kernel in `ap_ctrl_none` control mode, which is currently not fully supported by Vitis HLS. To overcome the limitation, we exported the HLS implementation as a Vivado IP, created an RTL wrapper, and exported the kernel as a `.xo` file. The HLS service is then instantiated by the RTL, and control signals (`ap_start`, `ap_done`, etc.) are ignored.

3.2 Detailed Architecture

Internally, the architecture is decomposed into independent and parallel read (*Request Processor*) and write (*Response Processor*) flows to maximise throughput (Figure 4). The *Short-Circuit Response* and *Error Handler* manage early responses when there is no need of contacting the application.

[6] https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html

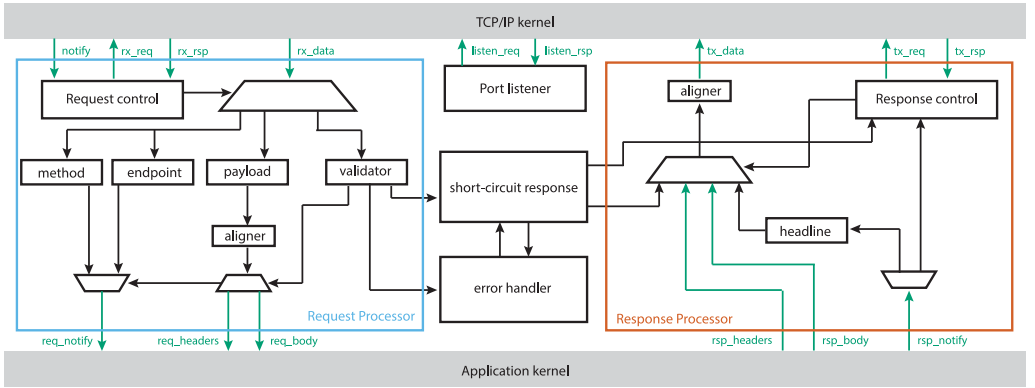

Fig. 4. Block diagram of the internal micro-architecture of Strega. Green arrows interfacing other kernels consist of AXI4-Stream uni-directional channels.

Even though the HTTP protocol can be seen in a pipeline fashion, our kernel supports asynchronous, non-blocking requests and responses. Applications can optimise scheduling and execution to best fit their requirements, allowing, e.g., responses to be issued in a different order as their respective request. For best performance and effective programming, the major part of components were implemented as a finite state machine inside a *dataflow* region. HLS stream structures are used for inter-module synchronisation.

No external memory access is required by Strega. It only uses ROM and BRAM internally to store HTTP constants and text templates. If necessary, then DDR instantiation is handled by the application. Applications that can process incoming HTTP requests at the maximal throughput do not need to allocate DDR resources for buffering, reducing resource utilisation overhead to a minimum. The data stream interfaces are 512 bit wide to match the word throughput of the TCP/IP stack.

The *Port Listener* is responsible for requesting to the TCP/IP kernel a port to be opened for listening to HTTP requests. In the current implementation, for simplicity, Strega listens to a single port, fixed at hardware compile time. This constraint can be easily lifted by allowing the application to communicate ports to be opened at runtime, given that the TCP/IP stack can listen to a very large number of ports dynamically [60].

The *Request Processor* is responsible of parsing, validating, and encoding incoming data. Based on the HTTP headline, it parses the endpoint and method of the request. If valid, then it first notifies the application of the incoming request, sending the method, endpoint, and the session ID that identifies the client, which should be used for the response. It then forwards HTTP headers and body payload to the application in two different streams. By splitting headers and body, the kernel allows for easy disposal of headers when they are not needed by the application. Unlikely in EasyNet, the communication primitives used in Strega do not know in advance how much data the engine will receive or send, so meta information coming from the TCP/IP layer, as well as the `last` flag of streams are used to assert the end of a message.

The lifetime of an HTTP request is as follows:

(1) The TCP/IP kernel sends a `notify` message to Strega with length and client information (destination port, IP address and session ID);
(2) Strega and the TCP/IP kernel handshake for reading a number of packets via `rx_req` and `rx_rsp`;
(3) Strega reads incoming data from TCP/IP via `rx_data`;

(4) Strega parses the method, endpoint, splits headers and body from the request;
(5) Strega notifies the application via `req_notify`;
(6) The application reads headers and body via `req_headers` and `req_body`.

The *Response Processor* listens to the application in a dedicated channel independent of the request flow. Upon notification, it parses the status code to compose the response's headline. The engine then merges application and server headers, which are followed by the body payload. For convenience of usage, Strega can merge unaligned data coming from the application. To issue the response, the engine forwards the total payload stream to the TCP/IP kernel together with the session ID that originated the request.

The lifetime of an HTTP response is as follows:

(1) The application notifies Strega of a response, providing status code, header and body lengths, and the session ID of the request;
(2) Strega creates the response headline based on the status code, and merges headers and body into a single, aligned stream;
(3) Strega and the TCP/IP kernel handshake for sending a number of packets via `tx_req` and `tx_rsp`;
(4) Strega sends outgoing data to TCP/IP via `tx_data`.

The *Short-circuit response* and the *Error Handler* modules are responsible for detecting and managing responses that can be directly issued without further processing, i.e., when Strega can issue the response without contacting the application. This can happen, for instance, when the request is malformed, or in cases the client application queries the HTTP server to pre-validate further requests, and the server answers with a `100 Continue` HTTP status. These two modules remove the need for the application to understand the HTTP protocol, so only valid and relevant requests reach the application layer.

3.3 Provision and Deployment

The period from when a cloud application is provisioned and instantiated until it is ready to operate is known as the *warming up period* [22, 56], which occurs, e.g., during scaling-out processes, releasing new service versions/updates, and virtualisation setup for container and serverless applications. Minimising the warming up period is important not only to save deployment costs, but also to quickly react to spike workload demands in scaling-out processes, or to reduce latency in serverless invocations. For FPGAs, the warming up period comprises programming the FPGA with the desired bitstream, memory manipulation such as allocation and data transfers to the device, and kernel invocation when they are not free-running kernels.

For maximal flexibility, the kernel running Strega does not require any memory-mapped registers, such as AXI4-Lite registers implemented by default in the Vitis Kernel Flow. This not only grants the kernel flexibility to be integrate with different FPGA shells (such as Coyote [32]), but also removes any deployment latency during warming up that would require a synchronisation with the host application. As soon as the FPGA is programed with the bitstream, Strega is warmed up and is able to open a TCP port and start listening for HTTP requests.

Using the system depicted in Figure 2 as the reference, the role of the host CPU is reduced to programming the FPGA with the corresponding bitstream, a process that takes at most 3 s. Applications that want to reduce the involvement of the CPU can also be deployed as either a free-running kernel, or remove the need of synchronous and recurrent FPGA invocation from the host. Section 4.1 details three applications created for the experimental evaluation of Strega. Two of them (*health check* and *to_uppercase*) uses a free-running kernel and is online as soon as the board is programmed. The third one (*static pages*) has a bit longer warming up, when

the CPU programs the board and launches the kernels, allocating memory and pushing data to the FPGA. Then, the application is online and does not require any further host intervention. By splitting control plane and data path (Figure 2), Strega contributes to restricting the intervention from the CPU during warming up only, while allowing kernels to be invoked via the network.

3.4 Development Support

As an open source project, we want to foster contributions from the community to all different levels of the stack. Equally important, developers willing to employ Strega should spend as little integration effort as possible, and get in return a solid server that completely abstracts away the HTTP and network details. For that purpose, we borrow some development principles from software engineering to design, code, test, evaluate, and iterate over applications using our platform.

First, we heavily integrated CMake for hardware integration and generation scripting, which largely reduces boilerplate code that is typically duplicated in several instances of projects. We extended CMake integration of IPs, kernels and program to not only Strega project, but also made contributions to the open source repositories that Strega is build on top of. To achieve this, we extended HLSlib [11] to support not only Vitis HLS but also Vivado resources. Two new function were added: `add_vitis_ip` and `add_vivado_kernel`. The first exports an HLS function as an IP that can then be instantiated in RTL by Vivado; the second packages a Vivado RTL design to a `.xo` Vitis kernel that can be later used to compose a program bitstream.

In the testing and debugging front, we leveraged the methods available from Vitis tools. We implemented a unit test (i.e., testbench) for each individual module of our kernel. The execution of unit tests happens via *C simulation* and validates the business logic of each module standalone. It is a software simulation, thus compilation, execution and debugging can occur at fast iteration steps, and OS I/O interfaces, such as `printf`, are available. Once the software logic is validated, *C-RTL co-simulation* synthesises the HLS code into RTL (i.e., Verilog and VHDL). A C wrapper instantiates the kernel, provides input mock data, consumes the output and validates it according to the expected values. This testing step helps identify misconceptions when coding hardware using a high-level language as C or C++.

Unfortunately, the final hardware implementation that is placed in the FPGA is still different from the one used for co-simulation. To validate this piece, one needs to do *hardware emulation*. In this step, the full FPGA system composed of kernels, memory and role shell is synthesised and emulated. Since the TCP/IP layer is connected to network transceivers of the board, and currently there is no way of stimulating them on the hardware emulator, we created a mock version of the TCP/IP channel in HLS. It generates mock TCP/IP notifications and incoming messages, consumes outgoing data, and respects the behaviour of handshakes of `rx` and `tx` ports. With this mock module, we create a kernel that replaces the real TCP/IP layer in the *hardware emulation* bitstream. This is a common practice for end-to-end testing CD pipelines for complex systems in the software world.

4 EXPERIMENTAL EVALUATION

In this section, we present performance comparisons between our FPGA-based HTTP server and several setups exposing the kernel application via a CPU-based HTTP frontend, and as an OpenCL function over the PCIe bus. We also report measurements from instrumented hardware experiments that show, with high precision, the latency imposed by the server in request and response processing.

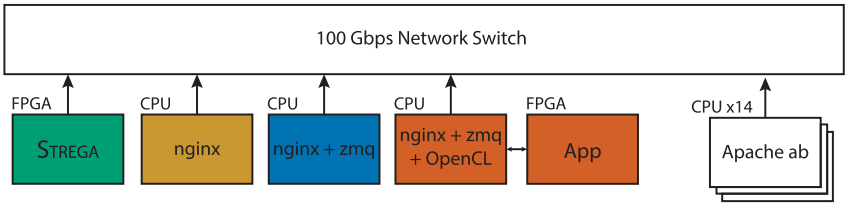

Fig. 5. Experimental setup. All nodes are connected to the same top-of-rack switch. Up to 14 CPU HTTP clients (Apache ab benchmarking tool) are used to produce enough workload to saturate STREGA.

4.1 Experimental Setup

For our experiments, we built STREGA using Vitis 2022.2 and deployed it on an Alveo U55C board.[7] The baselines were executed on AMD EPYC 7302P servers with 32 vCPU and 64 GiB of memory each. Up to 14 CPU servers were used to act as clients and issue HTTP requests to all server deployments. Figure 5 shows the setup, all HTTP servers and clients are placed in the same 100 Gbps network, routed through the same top-of-rack network switch, a Cisco Nexus 9336c FX2, using optical cables, thus minimising network disturbances.

Applications: We created three simple web applications on the FPGA: (i) a health check, similar to what is used in CPU load balancers, that responds with a body payload in `json` indicating if the FPGA design is running healthy. This application has close to zero business logic other than respecting the consuming and producing flows of STREGA, thus the relevance to measure the strict minimum latency imposed by the HTTP server; (ii) a text transformer *to_uppercase* that receives plain text and converts characters `[a-z]` to `[A-Z]`, returning the same amount of data as the input; and (iii) a simple static server that serves objects directly from the FPGA memory to the internet via `GET` requests. The latency and throughput of DDR memory access, in this example, tends to represent generic kernel characteristics running on the FPGA. The goal of the evaluation is not to measure the acceleration of the applications running on top of STREGA but rather the HTTP server itself. Despite this, these applications serve the purpose of showing the feasibility of the solution in terms of both flexibility and ease of integration.

Acting as baselines, we compare STREGA to two other methods of invoking an FPGA kernel. First, we designed an OpenCL application running on the host CPU for a traditional invocation over the PCIe bus. Second, given the HTTP interface brought by STREGA, we built a hybrid deployment, where an HTTP server running on the host CPU acts as the frontend of the kernel, and invokes the application through the PCIe bus using OpenCL. For each of the baselines, we also extend the evaluation to measure the performance of individual elements, i.e., the latency decomposition of an OpenCL invocation into data transfers and kernel execution, as well as the latency impact of each software element in the hybrid solution.

PCIe baseline: We designed an OpenCL host application following the best optimisations to reduce latency and increase throughput for a traditional FPGA-based kernel invocation over the PCIe bus. These optimisations can be found as the `host/overlap` and `host/data_transfer` demos from AMD's Vitis Accel Examples GitHub repository.[8]

Hybrid baseline: Acting as the CPU-based HTTP frontend, we deployed nginx open source,[9] an HTTP server tuned for scalability and high performance. It presents an increasing adoption encompassing diverse use cases and architectures, being used by Netflix [43], Dropbox [29], and

[7] https://www.xilinx.com/products/boards-and-kits/alveo/u55c.html
[8] https://github.com/Xilinx/Vitis_Accel_Examples/tree/master/host
[9] https://nginx.org

WordPress [45]. For a fair comparison, we built nginx from source, disabling all advanced and optional modules and features not yet implemented in Strega, such as body gzip/brotli compression, logs, SSL, and any sort of authentication or rate limit. Table 3 in Appendix A lists all the flags used in these evaluations. This being said, it is important to stress that it is not possible to downgrade nginx to have the exact same set of features and HTTP protocol implementation as Strega, leave aside numerous third-party modules developed across years of utilisation. This is also a point that should be noticed for FPGA-based infrastructure: by making Strega open source, we hope more developers will contribute to the ecosystem and extend it further. We allocate all 32 cores available in the server machine to spawn nginx processes, and set `worker_rlimit_nofile` to 8192, which dictates the maximum number of simultaneous connections that can be opened by an nginx process. These two parameters consist of much higher values than used in production environments, as they allocate the whole computing power of the CPU to the HTTP server, leaving no space left for main backend applications. Additionally, we provide the body payload of the *health-check* endpoint directly in nginx configuration file, so no disk access or external application process is required. Connecting the HTTP server to the backend application, we use ZeroMQ,[10] an efficient messaging library for concurrency communication between applications.

4.2 Benchmark Tool

To issue client requests, we used standard software HTTP clients, which also ensures the correctness of the implementation and its compliance with the protocol. For throughput measurements, we first need an HTTP benchmark client that allows stressing the server with numerous parallel requests issued from multiple clients. The Apache HTTP server benchmarking tool (`ab`)[11] provides the right flexibility in terms of request customisation and profiling metrics: It is possible to issue any HTTP method, any body payload size and define any custom header. However, `ab` is found alongside several other tools from Apache software, and since they target measurements of production environments and client measurements over several layers of network, results are rounded up to milliseconds. For our experiments, we are interested in measuring the intrinsic metrics of Strega and nginx, reducing network interference as much as possible. We modified `ab` to preserve measurement numbers at microsecond granularity, and for easier benchmark execution, we also prepared a standalone compilation and installation of the tool, which we published as an open source repository[12] that can be reused in similar use cases.

4.3 Intrinsic Latency

We instrumented Strega with ChipScopes [67] that capture, at clock-cycle granularity, the hardware signals of AXI4-Stream interfaces of the system while running on the physical board. We used these measurements to compute the processing time of different tasks done in Strega to establish the intrinsic latency provoked exclusively by its implementation alone. In other words, we isolated and measured the latency introduced by Strega to the overall system. Given that adding the ChipScope to the design considerably reduces the clock frequency of the placed and routed system, we convert measurements into clock cycles and report the key ones also in nano seconds, taking the clock frequency of a standard design as the reference, i.e., 254 MHz.

From the moment when the TCP/IP layer sends an incoming packet notification, until Strega issues the request notification to the application, there are 135 clock cycles, so 531 ns, as illustrated in Figure 6. This latency is greatly concentrated by the parsing of HTTP headline, as the handshake

[10] https://zeromq.org/
[11] https://httpd.apache.org/docs/2.4/programs/ab.html
[12] Available at https://github.com/fpgasystems

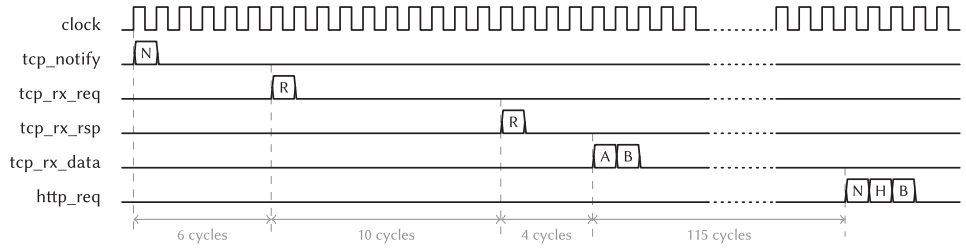

Fig. 6. Waveform of the request flow as captured by a ChipScope.

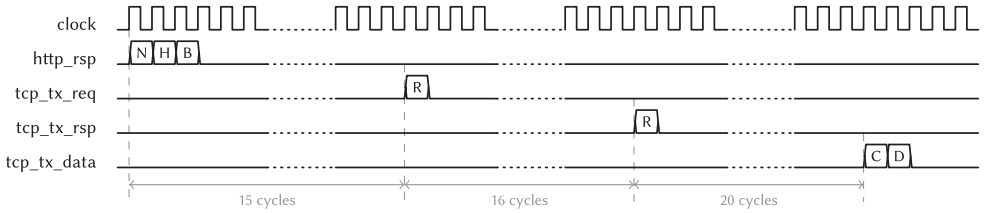

Fig. 7. Waveform of the response flow as captured by a ChipScope.

with the TCP/IP stack takes only 16 cycles, and the first data line arrives four cycles later. Once the headline is parsed, the application is notified and the HTTP server simply forwards the payload. Each 512 bit line of request payload takes 6 clock cycles to be streamed from the TCP/IP layer to the application.

In the transmission flow, the generation of the HTTP response headline is hidden by processing it in parallel to the TCP/IP transmission handshake, which greatly impacts the overall latency. From the moment when the application notifies Strega with a response message, until it starts sending data to the TCP/IP layer, it takes 51 clock cycles, or 201 ns, less than a half compared to the request flow, as illustrated in Figure 7. Once a steady stream of data is established to forward headers and response body to the TCP/IP layer, Strega can produce a 512 bit line every 8 clock cycles. The difference with the request flow for this metric is provoked by the necessity of alignment between the data sent by the application and the one send to the TCP/IP layer. Regardless of this difference, the transmission flow can still sustain enough throughput to process data at line-rate speed as well.

4.4 End-to-end Latency

In this experiment, we place a single HTTP client running on a CPU to issue sequential requests to both Strega and the hybrid baseline. Additionally, we also compare the same FPGA function invocation as a traditional OpenCL call, thus going through the PCIe bus.

For the former measurement, we measure the total completion time, encompassing the entire network round trip from the CPU-based client, to one of the HTTP servers, and back to the client. As for the latter measurement, we measure only the data transfer from the CPU to the FPGA device, the kernel execution, and the retrieval of the result data to the host memory after kernel completion. In both cases, the request and response payloads consist of 64 B, and the deployed application across different servers is the `to_uppercase` function.

Figure 8 shows the latency distribution of 5,000 function invocations for each method. Maximum, minimum, 95th and 5th percentiles are also reported. For better visibility, y-axis is in logarithmic scale. Strega shows a consistent lower latency compared to any other baseline setup, as low as 16 µs compared to 62 and 304 µs for *PCIe* and *hybrid* respectively. An interesting point to highlight comes from the sample variance, or lack thereof: Only 4 of 5,000 measurements in Strega have

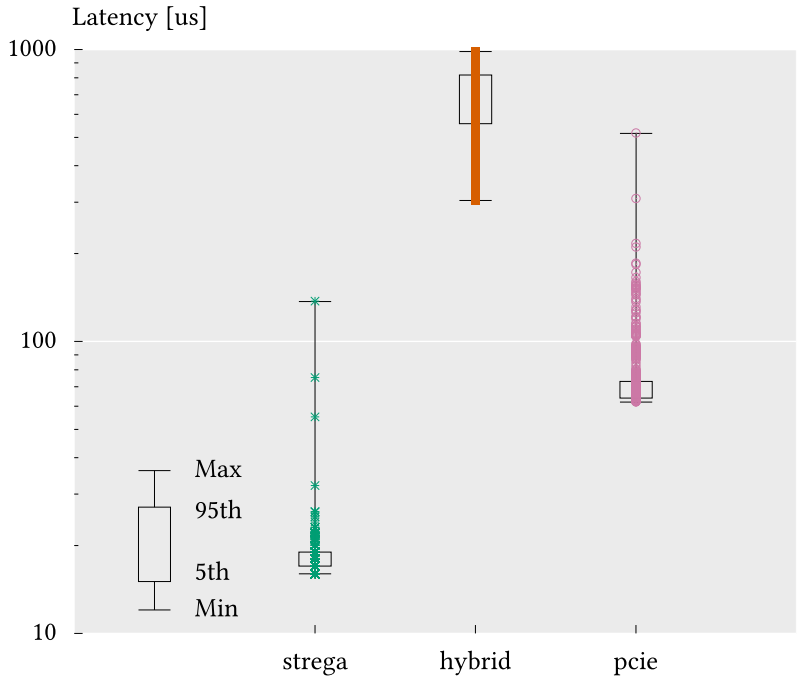

Fig. 8. Latency distribution of 5,000 sequential invocations as HTTP request-response transactions issued from a single CPU client for Strega and *hybrid*, and as OpenCL kernel calls for *PCIe*.

latency higher than 30 µs, and the gap between the minimum and the 95th percentile is only 3 µs, while for *PCIe* and *hybrid* this gap represents 11 and 514 µs, respectively. Strega also presents the smallest tail latency, the difference between the maximum value and the 95th percentile is 118 µs while for PCIe it reaches 443 µs.

The slowest point for the HTTP experiments (Strega and *hybrid*) is caused by the very first request of each experiment that needs to go over TCP handshake to establish the TCP session that will be reused in subsequent requests. Apart from this point, any other measurement of Strega is strictly faster or within the 95th percentile than any other of the baseline samples. This also highlights the importance of the advantages that the TCP/IP implementation in hardware brings to the overall system.

The fact that the end-to-end latency of invoking an FPGA-based kernel as an HTTP call over the network is faster than a traditional PCIe invocation is remarkable. The latency distribution plotted in Figure 8 shed light on the actual imbalance between what the FPGA hardware can achieve and what is often limited by the interconnect used, which has traditionally being the PCIe bus. We further discuss the impact of these numbers in Section 5.

4.4.1 HTTP Baselines. The *hybrid* baseline is composed of different elements, so we also perform different setups to characterise the impact of each one of them into the overall latency of the design. The first consists of *nginx* alone, where the payload of the response is statically defined in its configuration file, so we can measure the fastest, even if not usable, latency of the engine. The second, *ZeroMQ* comprises nginx, the ZeroMQ module, and a CPU implementation of the `to_uppercase` application, which mainly shows the latency introduced by the asynchronous communication framework. We also report the previous numbers from Strega and *hybrid* for comparison.

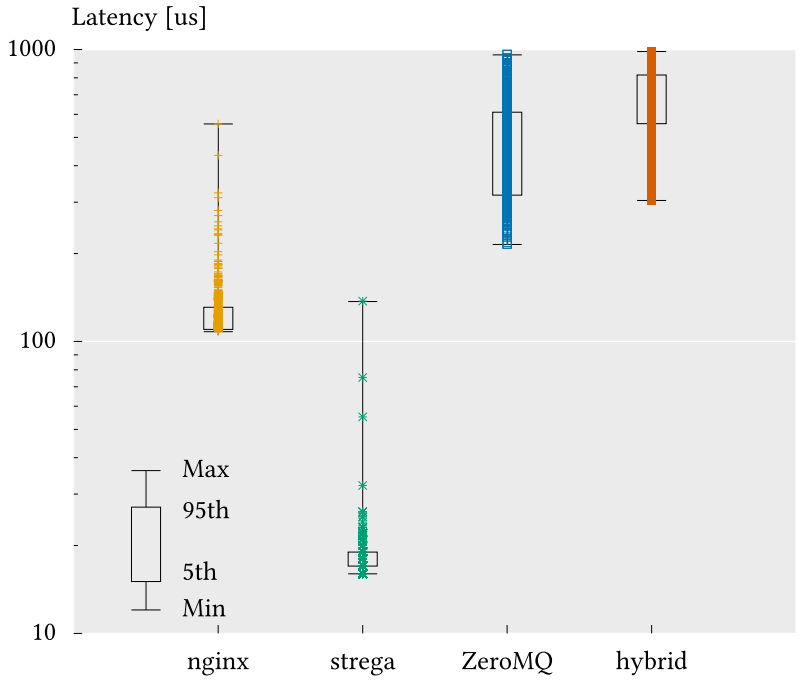

Fig. 9. Latency distribution of 5,000 sequential HTTP request-response transactions i ssued from a single CPU client. A single TCP/IP connection session is used for each experiment.

The latency measurements depicted in Figure 9 demonstrates that the ZeroMQ communication framework not only introduces significant latency, but also amplifies the discrepancy between the 5th and 95th percentiles, thereby diminishing the determinism of the call. Comparing the performance of the ZeroMQ baseline and the hybrid solution, we observe the additional latency pattern associated with PCIe communication, as ploted in Figure 8. These measurements underscore the importance of integration efforts in the middleware layer, specifically in connecting an HTTP server with the backend application. Previous research [49] has explored similar integration setup utilising uWSGI[13] as the middleware, but with inferior performance (in the millisecond range).

4.4.2 TCP/IP Breakdown. The global latency of a single HTTP request-response trip seen by the remote client is composed of the following components:

- *DNS lookup*: the time it takes to translate the domain name of the target server to its IP address. Since we are not interested in this metric, we address the servers using their IPs directly, so no external DNS lookup is necessary. There is still, however, a local check made by the client, so we report this metric;
- *TCP connect*: the time it takes to establish the three-way handshake of a TCP/IP connection. For this experiment, we force opening a new TCP/IP session for each request, so this metric is captured at its highest value;
- *Client setup*: the time it takes for the client to issue the request;
- ***Time to first byte (TTFB)***: once the request has been issued, how long the client is idle waiting for the response;

[13] https://uwsgi-docs.readthedocs.io

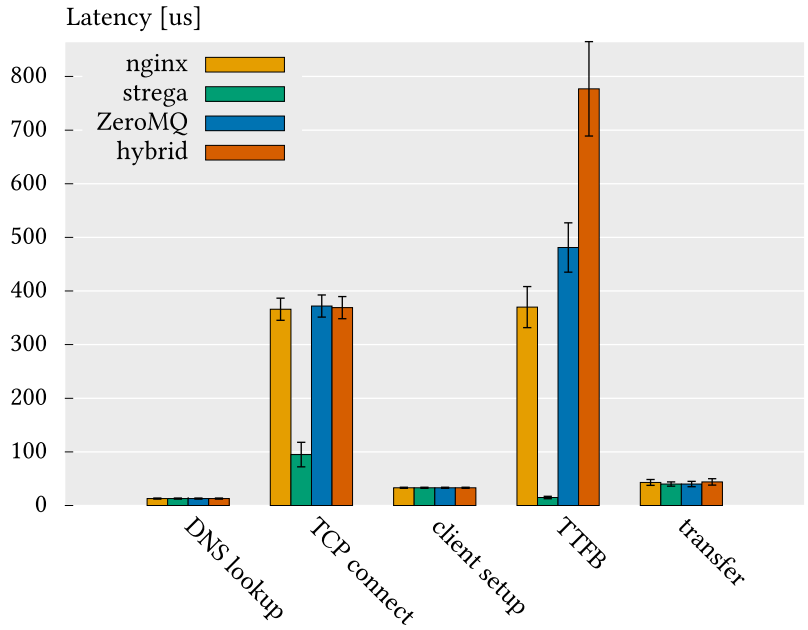

Fig. 10. Breakdown of the latency perceived by a CPU HTTP client. In this experiment, a TCP/IP session is created for each HTTP request.

- *Data transfer*: once the first byte is received, how long it takes to receive the full response.

In addition to these components, a client may also need to handle HTTP redirections when the server provides a response indicating that the client should make another request to a different endpoint to complete the initial request. When using HTTPS, the client must also complete an SSL handshake for each request made, which results in additional latency. These two metrics were neutralised in all of the experiments in this article as they do not depend on the HTTP server.

Figure 10 presents the breakdown of the latency perceived by a CPU client from the moment an HTTP request is issued until the reception of its response. The TCP/IP handshake between the CPU client and the CPU baselines takes about 366 µs, whereas the offloaded TCP/IP stack to the FPGA reduces the operation to 95 µs. It is important to highlight that using a CPU to issue the client request induces higher latency than compared to a FPGA-FPGA communication [13, 25], so the TCP/IP connect metric would be even faster if the HTTP client was placed on an FPGA. The TTFB captures the real latency imposed by the servers modulo the network delay, which in our setup is irrelevant. Nginx presents a latency of 370 µs, almost 25 times higher than the TTFB of Strega, which is not only faster but also presents a very deterministic value, compared to the big variance of nginx. The addition of ZeroMQ increases the TTFB in 110 µs compared to the pure-nginx solution. The FPGA invocation on top of ZeroMQ increases the TTFB in further 296 µs. The transfer time of the 16 B of health check payload is characterised by the network topology rather than the server stack, so both servers perform similarly. Since DNS lookup and client setup are client-dependent elements, there is no difference between Strega and nginx. Also, since IP addresses were used and no disk reads were required, both metrics present close to zero variance.

HTTP/1.1 provides the ability for a client to reuse an existing TCP/IP session when issuing subsequent requests, which considerably reduces the latency of subsequent transactions by eliminating the three-way connection handshake. Strega supports this feature by default and

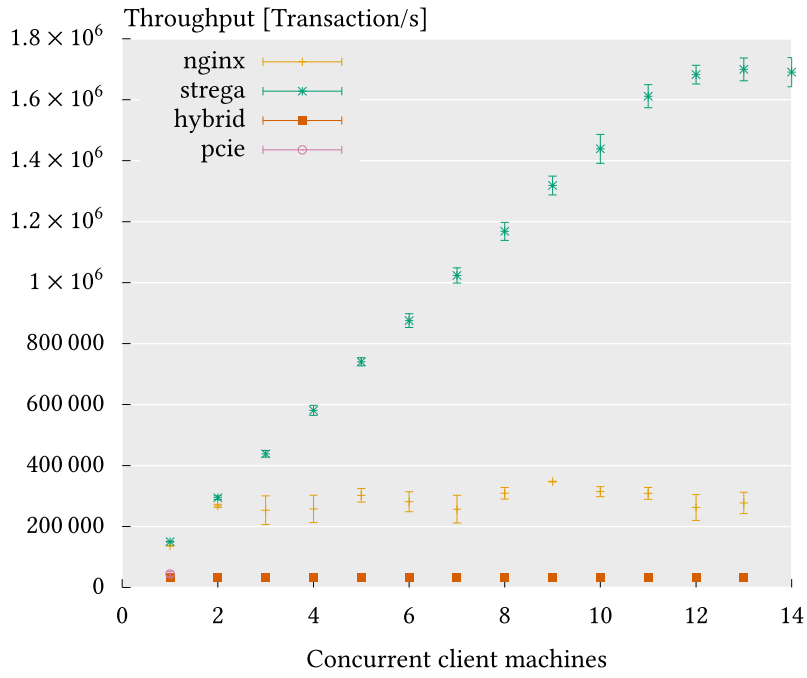

Fig. 11. Stress-test of Strega, nginx, and hybrid setup to saturate the servers in terms of transactions processed per second. Up to 14 CPU clients, and 64 threads per machine were allocated for the experiments.

preserves the TCP connection open after a response is sent back to the client. To test the equivalent behaviour in nginx, we add the `Connection: Keep-alive` header to the requests. For equivalence in the headers size, we add the same header to requests to Strega, even if they could have been omitted.

4.5 Throughput

In additional to latency, another key performance number for an HTTP server is the number of request-response transactions that can be processed per second. To be able to produce enough workload in our experiments, we adjust the concurrency level of our benchmarking tool to the double of vCPUs available in the machine, i.e., there are 64 threads running on the CPU client issuing HTTP requests in parallel. We issue about 10 million requests per experiment, which takes several minutes to complete, saturating the network and thus reducing variance in the measurements. In total, we deploy all 14 available machines in our cluster for this experiment. For the CPU baselines, only 13 CPU clients are available, since one of the machines is used as HTTP server.

For the PCIe baseline, we ensure that the execution of the kernel and the communication over the PCIe bus are fully saturated by providing unlimited workload to the OpenCL queues responsible for scheduling the kernel execution and awaiting its completion. The workload for this series can only be generated from a single CPU client, which corresponds to the host CPU itself (i.e., the *hybrid* baseline provides numbers for a multi-server setup over the PCIe).

Figure 11 plots the stress test of Strega, nginx, and hybrid in terms of transactions per second. Note that the PCIe measurement, only possible from a single client machine, is slightly hidden behind the hybrid measurement. Nginx is able to process up to 370,000 transactions per second,

Table 1. Resource Utilisation and Maximum Clock Frequency on Alveo U55C

Interface	LUTs		Registers		BlockRAM		Clock
OpenCL	124,025	(9.51 %)	167,496	(6.42)	199	(9.87 %)	500 MHz
TCP/IP [25]	132,742	(10.2 %)	—	—	522	(25.9 %)	254 MHz
Strega	272,096	(20.8 %)	430,244	(16.5 %)	401	(19.9 %)	254 MHz
Alveo U55C	1,303,680	(100 %)	2,607,360	(100 %)	2016	(100 %)	500 MHz

Table 2. Overall Comparison of Different Interfaces for FPGA-based Kernels

Interface	Latency	Throughput	Client stack	API	Scalability
PCIe	64 µs	44,000 T/s	custom driver	OpenCL	single client
Hybrid	557 µs	32,000 T/s	generic HTTP	full HTTP	very flexible
Strega	17 µs	1,700,000 T/s	generic HTTP	limited HTTP	very flexible
TCP/IP [25]	4.3 µs	—	TCP	low-level TCP	very flexible

which is the load from three client machines. The hybrid baseline suffers from the already limited throughput from nginx, but also from the PCIe bus, that cannot issue more than 44,000 OpenCL calls per second. A single CPU client machine is able to issue about 150,000 requests per second, and at least 11 of them are required to saturate a single node of Strega, at around 1,700,000 transactions per second, close to five times higher than what a single *nginx-alive* node can process.

Since the client application has close to zero business logic as fully utilises all the available cores to simply issue requests and receive responses, in a real deployment, the number of client machines required to saturate Strega is necessarily bigger than the one presented in Figure 11.

The performance numbers, 1.7 M transactions per second, 16 µs of latency—5 times higher and one order of magnitude faster than a commercial 32-core CPU baseline—show that placing a CPU in front an FPGA for acceleration will very likely limit the acceleration that a decoupled FPGA device would be able to achieve otherwise, which is the behaviour mocked in the hybrid baseline. The traditional architecture, where requests arrive by network to the CPU and are then transferred to the FPGA is therefore only interesting in cases where throughput is not an issue, or in latency-insensitive use cases.

4.6 Hardware Utilisation

Table 1 shows the resources consumed by Strega after place and route on an Alveo U55C board. This utilisation comprises the full bitstream, including the static XRT shell, the `to_uppercase` application, the TCP/IP stack and memory controllers. For a fair comparison, we also report the resource consumption of the same application in the pure OpenCL model. Notice that the actual board utilisation is not necessarily linear to the actual functionality routed in the board. Removing the OpenCL interface from the simple `to_uppercase` and adding Strega (with all the TCP/IP requirements) increases the overall resource utilisation by about 10 %. There is a considerable overhead for small circuits. The clock frequency of Strega, 254 MHz is set by the TCP/IP stack. Note that more than 80% of the board is available to the application, plenty to run substantial microservices.

4.7 Comparison

Table 2 summarises the comparison elements of the FPGA-based kernel interfaces analysed in this article. We compare the HTTP interface of Strega with the traditional PCIe model, as well as a the hybrid baseline. We also report results from Reference [25], where the kernel is also available over the network, but with a lower-level protocol.

5 DISCUSSION

More than an HTTP server architecture and implementation, the deployment of Strega in FPGA-based kernels for data centre applications sheds light on architectural decisions that were taken for granted. As heterogeneous computing becomes more accessible and affordable, external tasks that were usually performed on a CPU can be offloaded to a more appropriated device.

An alternative to the PCIe bus. The utilisation of the OpenCL programming model for communication between the host CPU and FPGA accelerator via the PCIe bus was originally borrowed from the GPU ecosystem. This approach facilitated the rapid integration of FPGA kernels into existing software and communication patterns. However, it is important to note that the OpenCL programming model was initially designed to leverage the acceleration characteristics of GPUs, which by essence involve processing large volumes of data [37, 70]. In contrast, FPGAs do not necessarily follow the same principle, and can operate efficiently as fine-grained dataflow units, handling smaller data sets at a time. The PCIe bus, as evident from both Figure 8 and Figure 11, does not perform optimally when the ratio between computation and data transfer is high, meaning that there is insufficient data to maximise its throughput. To overcome this limitation, the conventional approach has been to batch process multiple individual invocations into a single operation, which trades higher throughput for significantly increased individual latency [39]. The introduction of Strega, which provides an even higher level of abstraction for FPGA-based kernels compared to OpenCL without compromising performance, marks a significant milestone in the integration of heterogeneous hardware into distributed systems.

Decoupling client application from the FPGA-kernel stack. A lot of effort and time is needed to learn all the nuances and master the optimisation techniques needed to develop FPGA-based kernels and integrate them into a system. The platform abstraction brought by the HTTP layer to an FPGA kernel allows client applications to be completely unaware that the kernel is being executed on an FPGA. In fact, this is particularly valuable in refactoring tasks, where small parts of the system can be modified, evaluated and delivered in small modules. Just like a microservice that can be updated, upgraded and re-implemented without the need of modifying its external client applications, an FPGA kernel with an HTTP interface can be more easily integrated into a distributed system. In terms of design, kernel and applications can be optimised independently, by different developers with different skills, without raising technology stack conflicts. In terms of maintenance and deployment, host applications do not need to provide support to hardware-specific stacks, such as OpenCL, FPGA board drivers, and so on.

Reconsidering CPU host and cloud offerings. Once the application does not need to sit on the host CPU that is PCIe-attached to the FPGA board, one can reconsider the usage of such host. As presented in Section 4.5, an HTTP stack can be executed with better performance by an FPGA. Imposing requests coming from a distributed system to hop into the CPU to then be routed to the FPGA results in a large performance penalty. By standardising the interface for FPGA kernels in the context of heterogeneous cloud computing, cloud users would not use host CPUs for tasks other than provisioning and launching the FPGA bitstream. Cloud providers should provide specialised services for such tasks, in such a manner that FPGA boards could be rented without the need of a renting a fixed CPU. Financially, it would reduce the hourly costs of FPGA in the cloud, but also it would simplify their utilisation. Pemberton et al. [48] tackled this problem targeting GPU Kernel-as-a-Service by making the switch from non-shareable accelerators, to small functions decoupled from infrastructure. Similarly, Tarafdar et al. [64] presented a light-weight FPGA provisioning platform for the cloud, making the host a mere management core, handled by the cloud provider. Both works go in the direction of making FPGAs a first-class citizen in the cloud.

Serverless FPGA. Once the control plane and data path of FPGA-kernels are separated, the former taken care by cloud providers while the later available through the HTTP interface, the abstractions for provisioning FPGAs on-demand just like a serverless application can be envisaged. In fact, FPGAs have the advantage of being able to be partially reconfigured at runtime. For this purpose, we envisage to integrate Strega with Coyote [32] to provide fast reconfigurable capabilities, so that microservices can be easily provisioned on the FPGA, thereby not tying the accelerator to one particular application [40]. This is specially useful for serverless and short-lived applications where Coyote offers the possibility of deploying up to 10 different microservices on one FPGA, which can be dynamically exchanged for others independently and without having to the restart the FPGA. This aligns with the goals of serverless computing, as Schleier-Smith et al. [55] advocated in their work, which aims to offer ease of development and scalability, potentially at lower costs (or, at most, at the same price) compared to non-serverless deployments.

Limitations. It should be noted that **High Performance Computing (HPC)** applications may not benefit from Strega. The focus of HPC solutions is performance rather than interoperability, similarly to highly optimised code with intrinsic operations that may be difficult to write, read, or maintain but provides better performance on specific CPUs. Similarly, optimised PCIe-attached shells and low-level network protocols might yield better performance for HPC applications than using an HTTP server like Strega.

5.1 Extending This Work

Given the widespread deployment of HTTP servers, this piece of software has overtime being extended with several complex features. The current version of Strega provides a simplified, yet realistic and fully operational, version of an HTTP server that can be used by applications whose goal is to be available over a RESTful API but do not need to handle all the features and optimisations available in the HTTP protocol. We plan to develop the following features as part of future versions of Strega:

TLS for HTTPS. In response to the growing emphasis on network communication encryption, there has been a significant push toward securing most of the network traffic.[14] FPGA-based implementations of AES encryption have demonstrated their effectiveness in this context [10]. By offloading the TLS protocol to the FPGA for AES encryption and decryption, in conjunction with the TCP/IP stack, significant performance gains can be achieved compared to deploying an HTTP server on a CPU.

Support to HTTP/2.0. The recent version of HTTP improves, among others, performance in the case of several parallel requests coming from the same client. This is done by slicing HTTP messages into HTTP frames (a similar principle to TCP/IP packets of a single session), and managing them at the presentation layer. This would also open up the possibility of porting gRPC to FPGAs, since it is built on top of HTTP/2.0.

Payload compression. To reduce network traffic, modern engines can compress the body payload before sending to the client. This feature would seat inside the HTTP kernel, making compression transparent to the application. DEFLATE decompression, one of the compression algorithms used in HTTP body, has also already been ported to FPGAs [10].

Call orchestration. Instead of using the HTTP request-response model to send input data and get it back to client, adopting a linear dataflow of information processing could be an interesting

[14] for instance, https://www.cnet.com/tech/services-and-software/google-accelerates-encryption-project

and complementary research direction. In this scenario, the FPGA could receive the payload from a node, process it, and forward the result to a yet different node. Enabling the FPGA to issue HTTP requests, which is a relatively straightforward task, would facilitate the orchestration of dataflow, just like invocations of microservices that sequentially call each other to accomplish a specific task.

Vendor-agnostic stack. The current implementation is tied to a TCP/IP stack that has only been ported to Xilinx boards. In an effort of moving away from vendor-specific solutions, we plan to port the whole stack to Intel FPGA boards as well, increasing the hardware abstraction for future applications regardless of the underlying board.

6 CONCLUSION

As specialised hardware architectures continue to evolve, nourished by the cloud, the relationship between accelerators and the host (CPU) can be reevaluated. Rather than be dependant and require CPU intervention, FPGAs offer the opportunity of being directly connected to the internet.

We presented Strega, an open source HTTP server for FPGAs. By increasing the hardware abstraction in the communication between client applications and FPGA kernels, Strega exposes the kernel as a simple RESTful interface over the network. As a consequence, FPGAs can be more easily adopted as microservices, while preserving all the performance advantages of heterogeneous computing. The opportunity of such standardised interface should help making FPGAs first-class processing units in the cloud, removing the requirement of necessarily instantiating a CPU for each FPGA board.

Experimental results demonstrate that Strega can handle up to 1.7 M HTTP requests per second with an end-to-end latency of 16 µs, comparable to traditional PCIe invocation. These numbers not only supports the trend of decoupling the control plane from the data path but also sheds light on the scalability problem of imposing CPU hoping in FPGA cloud offerings.

APPENDIX

A NGINX DISABLED MODULES AND FEATURES

The following table lists all modules and feature flags from nginx disabled for the experimental evaluation.

Table 3. Resource Utilisation and Maximum Clock Frequency on Alveo U55C

Location	Flag
build	`--without-http_access_module`
build	`--without-http_auth_basic_module`
build	`--without-http_autoindex_module`
build	`--without-http_browser_module`
build	`--without-http_charset_module`
build	`--without-http_empty_gif_module`
build	`--without-http_fastcgi_module`
build	`--without-http_geo_module`
build	`--without-http_gzip_module`
build	`--without-http_limit_conn_module`
build	`--without-http_limit_req_module`
build	`--without-http_map_module`
build	`--without-http_memcached_module`
build	`--without-http_proxy_module`
build	`--without-http_referer_module`
build	`--without-http_scgi_module`
build	`--without-http_ssi_module`
build	`--without-http_split_clients_module`
build	`--without-http_upstream_hash_module`
build	`--without-http_upstream_ip_hash_module`
build	`--without-http_upstream_least_conn_module`
build	`--without-http_upstream_zone_module`
build	`--without-http_userid_module`
build	`--without-http_uwsgi_module`
build	`--without-http-cache`
build	`--without-mail_pop3_module`
build	`--without-mail_imap_module`
build	`--without-mail_smtp_module`
runtime	`proxy_redirect off;`
runtime	`access_log off;`
runtime	`worker_rlimit_nofile 8192;`

ACKNOWLEDGMENTS

We thank Zhenhao He for his help on debugging the system. We are grateful to AMD for the donation of the equipment used in this article. The research of Fabio Maschi has been supported in part by a donation from SAP. Strega is open source; it can be found together with the additional tools at https://github.com/fpgasystems/strega.

REFERENCES

[1] Yalemisew Abgaz, Andrew McCarren, Peter Elger, David Solan, Neil Lapuz, Marin Bivol, Glenn Jackson, Murat Yilmaz, Jim Buckley, and Paul Clarke. 2023. Decomposition of monolith applications into microservices architectures: A systematic review. *IEEE Trans. Softw. Eng.* (2023), 1–32. https://doi.org/10.1109/TSE.2023.3287297

[2] Nish Anil, Tarun Jain, John Parente, and Maira Wenzel. 2022. Microservices architecture: .NET microservices—Architecture e-book. Retrieved from https://learn.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/microservices-architecture

[3] Marco Bacis, Rolando Brondolin, and Marco D. Santambrogio. 2020. BlastFunction: An FPGA-as-a-service system for accelerated serverless computing. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE'20)*. IEEE, 852–857. https://doi.org/10.23919/DATE48585.2020.9116333

[4] Mike Belshe, Roberto Peon, and Martin Thomson. 2015. Hypertext Transfer Protocol Version 2 (HTTP/2). RFC 7540. https://doi.org/10.17487/RFC7540

[5] Florian Braun, John W. Lockwood, and Marcel Waldvogel. 2002. Protocol wrappers for layered network packet processing in reconfigurable hardware. *IEEE Micro* 22, 1 (2002), 66–74. https://doi.org/10.1109/40.988691

[6] Robert Brzoza-Woch and Piotr Nawrocki. 2016. FPGA-based web services—Infinite potential or a road to nowhere? *IEEE Internet Comput.* 20, 1 (2016), 44–51. https://doi.org/10.1109/MIC.2015.23

[7] Robert Brzoza-Woch, Andrzej Ruta, and Krzysztof Zielinski. 2013. Remotely reconfigurable hardware-software platform with web service interface for automated video surveillance. *J. Syst. Archit.* 59, 7 (2013), 376–388. https://doi.org/10.1016/j.sysarc.2013.05.007

[8] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink™: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.* 38, 4 (2015), 28–38. http://sites.computer.org/debull/A15dec/p28.pdf

[9] C. E. Chang, F. Mohd-Yasin, and A. K. Mustapha. 2009. An implementation of embedded RESTful Web services. In *Proceedings of the Innovative Technologies in Intelligent Systems and Industrial Applications*. 45–50. https://doi.org/10.1109/CITISIA.2009.5224244

[10] Monica Chiosa, Fabio Maschi, Ingo Müller, Gustavo Alonso, and Norman May. 2022. Hardware acceleration of compression and encryption in SAP HANA. *Proc. VLDB Endow.* 15, 12 (2022), 3277–3291.

[11] Johannes de Fine Licht and Torsten Hoefler. 2019. hlslib: Software engineering for hardware design. arXiv:1910.04436. Retrieved from http://arxiv.org/abs/1910.04436

[12] Thomas Erl, Benjamin Carlyle, Cesare Pautasso, and Raj Balasubramanian. 2013. *SOA with REST—Principles, Patterns and Constraints for Building Enterprise Solutions with REST*. Pearson Education.

[13] Nariman Eskandari, Naif Tarafdar, Daniel Ly-Ma, and Paul Chow. 2019. A modular heterogeneous stack for deploying FPGAs and CPUs in the data center. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'19)*. ACM, 262–271. https://doi.org/10.1145/3289602.3293909

[14] Mike Field. 2016. FPGA Webserver. Retrieved from https://github.com/hamsternz/FPGA_Webserver

[15] Roy Thomas Fielding. 2000. *Architectural Styles and the Design of Network-based Software Architectures*. Ph. D. Dissertation. University of California, Irvine, CA.

[16] Roy Thomas Fielding, Mark Nottingham, and Julian Reschke. 2022. HTTP Semantics. RFC 9110. https://doi.org/10.17487/RFC9110

[17] Roy Thomas Fielding, Mark Nottingham, and Julian Reschke. 2022. HTTP/1.1. RFC 9112. https://doi.org/10.17487/RFC9112

[18] Daniel Firestone, Andrew Putnam, Sambrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian M. Caulfield, Eric S. 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 G. Greenberg. 2018. Azure accelerated networking: SmartNICs in the public cloud. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)*. USENIX Association, 51–66.

[19] Brad Fitzpatrick. 2004. Distributed caching with memcached. *Linux J.* 2004, 124 (2004).

[20] Mario Flajslik and Mendel Rosenblum. 2013. Network interface design for low latency request-response protocols. In *Proceedings of the USENIX Annual Technical Conference*. USENIX Association, 333–346.

[21] 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 & edge systems. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*, Iris Bahar, Maurice Herlihy, Emmett Witchel, and Alvin R. Lebeck (Eds.). ACM, 3–18. https://doi.org/10.1145/3297858.3304013

[22] Google. 2023. Configuring Warmup Requests to Improve Performance. Retrieved from https://cloud.google.com/appengine/docs/legacy/standard/java/configuring-warmup-requests

[23] Google. 2023. What Is Microservices Architecture? Google Cloud Topics. Retreived from https://cloud.google.com/learn/what-is-microservices-architecture

[24] Michael Hausenblas and Jacques Nadeau. 2013. Apache drill: Interactive Ad-Hoc analysis at scale. *Big Data* 1, 2 (2013), 100–104. https://doi.org/10.1089/big.2013.0011

[25] Zhenhao He, Dario Korolija, and Gustavo Alonso. 2021. EasyNet: 100 Gbps network for HLS. In *Proceedings of the 31st International Conference on Field-Programmable Logic and Applications (FPL'21)*. IEEE, 197–203. https://doi.org/10.1109/FPL53798.2021.00040

[26] ISO. 1996. *Information technology–Open Systems Interconnection–Basic Reference Model: The Basic Model*. Standard. International Organization for Standardization, Geneva, CH.

[27] Zsolt István, David Sidler, and Gustavo Alonso. 2017. Caribou: Intelligent distributed storage. *Proc. VLDB Endow.* 10, 11 (2017), 1202–1213. https://doi.org/10.14778/3137628.3137632

[28] Ana Ivanchikj and Cesare Pautasso. 2020. Modeling microservice conversations with RESTalk. In *Microservices, Science and Engineering*. Springer, 129–146. https://doi.org/10.1007/978-3-030-31646-4_6

[29] Alexey Ivanov. 2017. Optimizing Web Servers for High Throughput and Low Latency. Retrieved from https://dropbox.tech/infrastructure/optimizing-web-servers-for-high-throughput-and-low-latency.

[30] Matthew Jacobsen, Dustin Richmond, Matthew Hogains, and Ryan Kastner. 2015. RIFFA 2.1: A reusable integration framework for FPGA accelerators. *ACM Trans. Reconfig. Technol. Syst.* 8, 4 (2015), 22:1–22:23. https://doi.org/10.1145/2815631

[31] Nivedita N. Joshi, P. K. Dakhole, and P. P. Zode. 2009. Embedded web server on Nios II embedded FPGA platform. In *Proceedings of the 2nd International Conference on Emerging Trends in Engineering & Technology (ICETET'09)*. IEEE Computer Society, 372–377. https://doi.org/10.1109/ICETET.2009.89

[32] Dario Korolija, Timothy Roscoe, and Gustavo Alonso. 2020. Do OS abstractions make sense on FPGAs? In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*. USENIX Association, 991–1010. https://www.usenix.org/conference/osdi20/presentation/roscoe

[33] Jay Kreps, Neha Narkhede, Jun Rao, et al. 2011. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, Vol. 11. 1–7.

[34] Julien Lallet, Andrea Enrici, and Anfel Saffar. 2018. FPGA-based system for the acceleration of cloud microservices. In *Proceedings of the IEEE International Symposium on Broadband Multimedia Systems and Broadcasting (BMSB'18)*. IEEE, 1–5. https://doi.org/10.1109/BMSB.2018.8436912

[35] 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)*. ACM, 36–51. https://doi.org/10.1145/3445814.3446696

[36] Ying Liu, Khaled Benkrid, Abdsamad Benkrid, and Server Kasap. 2009. An FPGA-based web server for high performance biological sequence alignment. In *Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems (AHS'09)*. IEEE Computer Society, 361–368. https://doi.org/10.1109/AHS.2009.59

[37] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2020. Pump up the volume: Processing large data on GPUs with fast interconnects. In *Proceedings of the International Conference on Management of Data (SIGMOD'20)*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 1633–1649. https://doi.org/10.1145/3318464.3389705

[38] Eduardo Magdaleno, Manuel Rodríguez, Fernando Pérez, David Hernández, and Enrique García. 2014. A FPGA embedded web server for remote monitoring and control of smart sensors networks. *Sensors* 14, 1 (2014), 416–430. https://doi.org/10.3390/s140100416

[39] Fabio Maschi and Gustavo Alonso. 2023. The difficult balance between modern hardware and conventional CPUs. In *Proceedings of the 19th International Workshop on Data Management on New Hardware (DaMoN'23)*, Norman May and Nesime Tatbul (Eds.). ACM, 53–62. https://doi.org/10.1145/3592980.3595314

[40] Fabio Maschi, Dario Korolija, and Gustavo Alonso. 2023. Serverless FPGA: Work-in-progress. In *Proceedings of the 1st Workshop on SErverless Systems, Applications and MEthodologies (SESAME'23)*, Dmitrii Ustiugov, Rodrigo Bruno, Pedro Fonseca, Boris Grot, and Antonio Barbalace (Eds.). ACM, 1–4. https://doi.org/10.1145/3592533.3592804

[41] Fabio Maschi, Muhsen Owaida, Gustavo Alonso, Matteo Casalino, and Anthony Hock-Koon. 2020. Making search engines faster by lowering the cost of querying business rules through FPGAs. In *Proceedings of the International Conference on Management of Data (SIGMOD'20)*. ACM, 2255–2270. https://doi.org/10.1145/3318464.3386133

[42] Norman May, Daniel Ritter, Andre Dossinger, Christian Färber, and Suleyman Demirsoy. 2023. DASH: Asynchronous hardware data processing services. In *Proceedings of the 13th Conference on Innovative Data Systems Research (CIDR'23)*.

[43] Netflix. 2023. Netflix Open Connect Appliances. Retrieved from https://openconnect.netflix.com/en/appliances/#software

[44] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. 2018. Understanding PCIe performance for end host networking. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM'18)*. ACM, 327–341. https://doi.org/10.1145/3230543.3230560

[45] nginx. 2012. NGINX at WordPress.com. Retrieved from https://www.nginx.com/success-stories/nginx-wordpress-com/.

[46] S Ojika, A Gordon-Ross, H Lam, B Patel, G Kaul, and J Strayer. 2018. Using FPGAs as Microservices: Technology, challenges and case study. In *Proceedings of the 9th Workshop on Big Data Benchmarks Performance, Optimization and Emerging Hardware (BPOE-9'18)*. https://par.nsf.gov/biblio/10073091

[47] Cesare Pautasso and Erik Wilde. 2010. RESTful web services: Principles, patterns, emerging technologies. In *Proceedings of the 19th International Conference on World Wide Web (WWW'10)*. ACM, 1359–1360. https://doi.org/10.1145/1772690.1772929

[48] Nathan Pemberton, Anton Zabreyko, Zhoujie Ding, Randy H. Katz, and Joseph Gonzalez. 2022. Kernel-as-a-service: A serverless interface to GPUs. arXiv:2212.08146. Retrieved from https://arxiv.org/abs/2212.08146

[49] Raphael Polig, Jagath Weerasinghe, and Christoph Hagleitner. 2017. RESTful web services on standalone disaggregated FPGAs. In *IEEE International Conference on Cloud Computing Technology and Science (CloudCom 2017)*. IEEE Computer Society, 114–121. https://doi.org/10.1109/CloudCom.2017.24

[50] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Logan Adams, Kypros Constantinides, John Demme, Daniel Firestone, Stephen Heil, Matt Humphrey, Daniel Lo, et al. [n. d.]. Retrospective: A reconfigurable fabric for accelerating large-scale datacenter services.

[51] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. 2020. FIRM: An intelligent fine-grained resource management framework for SLO-oriented microservices. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*. USENIX Association, 805–825.

[52] Burkhard Ringlein, François Abel, Dionysios Diamantopoulos, Beat Weiss, Christoph Hagleitner, Marc Reichenbach, and Dietmar Fey. 2021. A case for function-as-a-service with disaggregated FPGAs. In *Proceedings of the 14th IEEE International Conference on Cloud Computing (CLOUD 2021)*. IEEE, 333–344. https://doi.org/10.1109/CLOUD53861.2021.00047

[53] Mario Ruiz, David Sidler, Gustavo Sutter, Gustavo Alonso, and Sergio López-Buedo. 2019. Limago: An FPGA-based open-source 100 GbE TCP/IP stack. In *Proceedings of the 29th International Conference on Field Programmable Logic and Applications (FPL'19)*, Ioannis Sourdis, Christos-Savvas Bouganis, Carlos Álvarez, Leonel Antonio Toledo Díaz, Pedro Valero-Lara, and Xavier Martorell (Eds.). IEEE, 286–292. https://doi.org/10.1109/FPL.2019.00053

[54] Andrzej Ruta, Robert Brzoza-Woch, and Krzysztof Zielinski. 2012. On fast development of FPGA-based SOA services - machine vision case study. *Des. Autom. Embed. Syst.* 16, 1 (2012), 45–69. https://doi.org/10.1007/s10617-012-9084-z

[55] Johann Schleier-Smith, Vikram Sreekanti, Anurag Khandelwal, João Carreira, Neeraja Jayant Yadwadkar, Raluca Ada Popa, Joseph E. Gonzalez, Ion Stoica, and David A. Patterson. 2021. What serverless computing is and should become: The next phase of cloud computing. *Commun. ACM* 64, 5 (2021), 76–84. https://doi.org/10.1145/3406011

[56] Amazon Web Services. 2023. Amazon EC2 Auto Scaling, User Guide. Retrieved from https://docs.aws.amazon.com/autoscaling/ec2/userguide/ec2-auto-scaling-default-instance-warmup.html

[57] Amazon Web Services. 2023. Amazon EC2 Instance Types. Retrieved from https://aws.amazon.com/ec2/instance-types/

[58] Amazon Web Services. 2023. What Are Microservices? Retrieved from https://aws.amazon.com/microservices/

[59] Sunil Shukla, Neil W. Bergmann, and Jürgen Becker. 2008. A web server based edge detector implementation in FPGA. In *Proceedings of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI'08)*. IEEE Computer Society, 441–446. https://doi.org/10.1109/ISVLSI.2008.5

[60] David Sidler, Gustavo Alonso, Michaela Blott, Kimon Karras, Kees A. Vissers, and Raymond Carley. 2015. Scalable 10Gbps TCP/IP stack architecture for reconfigurable hardware. In *Proceedings of the 23rd IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'15)*. IEEE Computer Society, 36–43. https://doi.org/10.1109/FCCM.2015.12

[61] David Sidler, Zsolt István, and Gustavo Alonso. 2016. Low-latency TCP/IP stack for data center applications. In *Proceedings of the 26th International Conference on Field Programmable Logic and Applications (FPL'16)*. IEEE, 1–4. https://doi.org/10.1109/FPL.2016.7577319

[62] Akshitha Sriraman and Thomas F. Wenisch. 2018. μTune: Auto-tuned threading for OLDI microservices. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*, Andrea C. Arpaci-Dusseau and Geoff Voelker (Eds.). USENIX Association, 177–194.

[63] Naif Tarafdar, Nariman Eskandari, Thomas Lin, and Paul Chow. 2018. Designing for FPGAs in the cloud. *IEEE Des. Test* 35, 1 (2018), 23–29. https://doi.org/10.1109/MDAT.2017.2748393

[64] Naif Tarafdar, Thomas Lin, Eric Fukuda, Hadi Bannazadeh, Alberto Leon-Garcia, and Paul Chow. 2017. Enabling flexible network FPGA clusters in a heterogeneous cloud data center. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'17)*. ACM, 237–246.

[65] Jagath Weerasinghe, François Abel, Christoph Hagleitner, and Andreas Herkersdorf. 2015. Enabling FPGAs in hyperscale data centers. In *Proceedings of the IEEE 12th International Conference on Ubiquitous Intelligence and Computing and IEEE 12th International Conference on Autonomic and Trusted Computing and IEEE 15th International Conference on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom'15)*. IEEE Computer Society, 1078–1086. https://doi.org/10.1109/UIC-ATC-ScalCom-CBDCom-IoP.2015.199

[66] Jagath Weerasinghe, Raphael Polig, François Abel, and Christoph Hagleitner. 2016. Network-attached FPGAs for data center applications. In *Proceedings of the International Conference on Field-Programmable Technology (FPT'16)*. IEEE, 36–43. https://doi.org/10.1109/FPT.2016.7929186

[67] AMD Xilinx. 2022. Vitis Unified Software Platform Documentation: Application Acceleration Development (UG1393). Retrieved from https://docs.xilinx.com/r/en-US/ug1393-vitis-application-acceleration/Debugging-with-ChipScope

[68] Jibo Yu, Yongxin Zhu, Liang Xia, Meikang Qiu, Yuzhuo Fu, and Guoguang Rong. 2011. Grounding high efficiency cloud computing architecture: HW-SW co-design and implementation of a stand-alone web server on FPGA. In *Proceedings of the 4th International Conference on the Applications of Digital Information and Web Technologies (ICADIWT'11)*. 124–129. https://doi.org/10.1109/ICADIWT.2011.6041412

[69] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster computing with working sets. In *Proceedings of the 2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud'10)*. USENIX Association.

[70] Mark Zhao, Niket Agarwal, Aarti Basant, Bugra Gedik, Satadru Pan, Mustafa Ozdal, Rakesh Komuravelli, Jerry Pan, Tianshu Bao, Haowei Lu, Sundaram Narayanan, Jack Langman, Kevin Wilfong, Harsha Rastogi, Carole-Jean Wu, Christos Kozyrakis, and Parik Pol. 2022. Understanding data storage and ingestion for large-scale deep recommendation model training: Industrial product. In *Proceedings of the 49th Annual International Symposium on Computer Architecture (ISCA'22)*, Valentina Salapura, Mohamed Zahran, Fred Chong, and Lingjia Tang (Eds.). ACM, 1042–1057. https://doi.org/10.1145/3470496.3533044

[71] Olaf Zimmermann, Mirko Stocker, Daniel Lübke, Cesare Pautasso, and Uwe Zdun. 2019. Introduction to microservice API patterns (MAP). In *Proceedings of the 1st and 2nd International Conference on Microservices, (Microservices'17/'19), OASIcs, Vol. 78*. 4:1–4:17. https://doi.org/10.4230/OASIcs.Microservices.2017-2019.4

Received 14 February 2023; revised 7 June 2023; accepted 12 July 2023