Cerebros: Evading the RPC Tax in Datacenters

Arash Pourhabibi
arash.pourhabibi@epfl.ch
EcoCloud, EPFL
Switzerland

Mark Sutherland
mark.sutherland@epfl.ch
EcoCloud, EPFL
Switzerland

Alexandros Daglis
alexandros.daglis@cc.gatech.edu
Georgia Institute of Technology
USA

Babak Falsafi
babak.falsafi@epfl.ch
EcoCloud, EPFL
Switzerland

ABSTRACT

The emerging paradigm of microservices decomposes online services into fine-grained software modules frequently communicating over the datacenter network, often using Remote Procedure Calls (RPCs). Ongoing advancements in the network stack have exposed the RPC layer itself as a bottleneck, that we show accounts for 40–90% of a microservice's total execution cycles. We break down the underlying modules that comprise production RPC layers and demonstrate, based on prior evidence, that CPUs can only expect limited improvements for such tasks, mandating a shift to hardware to remove the RPC layer as a limiter of microservice performance. Although recently proposed accelerators can efficiently handle a portion of the RPC layer, their overall benefit is limited by unnecessary CPU involvement, which occurs because the accelerators are architected as co-processors under the CPU's control. Instead, we show that conclusively removing the RPC layer bottleneck requires all of the RPC layer's modules to be executed by a NIC-attached hardware accelerator. We introduce Cerebros, a dedicated RPC processor that executes the Apache Thrift RPC layer and acts as an intermediary stage between the NIC and the microservice running on the CPU. Our evaluation using the DeathStarBench microservice suite shows that Cerebros reduces the CPU cycles spent in the RPC layer by 37–64×, yielding a 1.8–14× reduction in total cycles expended per microservice request.

CCS CONCEPTS

• **Computer systems organization** → **Architectures**; • **Information systems** → *Information integration*; • **Software and its engineering** → Cloud computing.

KEYWORDS

Remote Procedure Calls, Hardware Accelerators, Microservices, Datacenters, Networked Systems

MICRO '21, October 18–22, 2021, Virtual Event, Greece
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8557-2/21/10…$15.00
https://doi.org/10.1145/3466752.3480055

ACM Reference Format:
Arash Pourhabibi, Mark Sutherland, Alexandros Daglis, and Babak Falsafi. 2021. Cerebros: Evading the RPC Tax in Datacenters. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21), October 18–22, 2021, Virtual Event, Greece.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3466752.3480055

1 INTRODUCTION

Today's online services are commonly developed and deployed using microservices, where the desired business logic is decomposed into independent software modules [24, 37]. The adoption of microservices promises significant improvements in scalability, development velocity, and programmer productivity and has therefore emerged as standard practice by large and small enterprises alike [10, 32, 55, 75]. As in other distributed systems, microservices rely on a communication layer, the most common being Remote Procedure Calls (RPCs) [24, 37, 70]. RPCs have emerged as the backbone of software at scale in datacenters, with cloud providers developing custom RPC layers that are used organization-wide (e.g., Google's gRPC [26] or Facebook's use of Thrift [4, 21]).

Continued advancements in datacenter networks, transport protocols, and systems software have begun to expose these RPC layers as bottlenecks for microservice performance. Deploying software in a datacenter with 100Gbps NICs [16, 49], bespoke network topologies [27, 67], and optimized protocol stacks [9, 15, 23, 29, 49, 56] mandates that the RPC layer must be able to operate at a matching rate. We find that when deployed with an optimized network stack, microservices spend 40–90% of their on-CPU time executing the RPC layer, as opposed to the application's business logic. The end result is a microservice that runs up to 10× slower than it should due to its dependence on an RPC layer.

Despite the importance and prevalence of RPCs in microservices, production RPC layers remain an inefficient workload for general-purpose CPUs. To demonstrate this inefficiency, we study the RPC layer and find that transforming data back and forth between application-readable and network formats represents ∼95% of the RPC layer. As prior work has shown that CPUs lag behind NIC line rates by orders of magnitude for such data transformations [60], CPU-based approaches are unable to mitigate the RPC layer bottleneck. However, the fact that the vast majority of the RPC layer's cost is attributed to these common data transformations indicates that hardware specialization for such tasks is in fact a feasible solution, because the hardware will be universally applicable to all microservices relying on RPCs.

In this work, we show that hardware that only accelerates data transformation is insufficient to solve the RPC bottleneck because the CPU must still be involved in the flow of incoming RPCs. The CPU is involved because it is the endpoint to which the NIC sends incoming packets and because current accelerator proposals leave the remainder of the RPC layer's functionality to the CPU. This design pattern means that the CPU must explicitly send work to the accelerator multiple times per RPC, incurring a considerable offload overhead each time. We show that offload overheads reduce a state-of-the-art accelerator's achievable speedup by 23×, compared to a streamlined design where the accelerator has a direct interface to the NIC and executes the whole RPC layer, removing the CPU from the critical path of RPC layer processing.

Our thesis is that offloading the *entire* RPC layer to a NIC-integrated "RPC processor" simultaneously eliminates offload overheads and allows cost-effective silicon provisioning. Supporting this idea are three key insights. First, despite the extreme diversity of microservices, there are three mature modules that underpin the RPC layer, two of which can be readily offloaded to a data transformation accelerator. Second, the primary source of offload inefficiency is the fact that the third—seemingly small—RPC module is still performed on the CPU. The resulting split in the RPC layer between software and hardware introduces excessive fine-grained CPU-accelerator offload overheads. Therefore, it is both necessary and sufficient to perform all three modules with unified hardware in order to remove the split in functionality and eliminate offload overheads. Third, deploying an RPC processor as a NIC rather than per-CPU-core extension results in 5× less silicon area and also enables a unique performance optimization opportunity: a novel dispatch policy that assigns RPCs to cores to increase instruction cache locality. Both of these improvements are enabled by the emergence of on-chip NIC integration [14, 57].

Building on these insights, we present Cerebros, an RPC processor that can execute production RPC layers such as Apache Thrift [21] in hardware, leaving the microservices' business logic alone to be executed on the CPU. Cerebros reduces the cycles spent in the RPC layer by 37–64× compared to software and up to 23× compared to prior accelerators [60]. Our work also boosts the performance of the microservice itself by improving the CPU's instruction fetch performance in two ways. First, offloading the RPC layer to Cerebros shrinks the microservice's working set by 27–68%, reducing the instruction cache's MPKI by up to 93%. Second, to address pathological microservices where the working set remains large even after offloading the RPC layer, Cerebros improves temporal locality by steering incoming requests to cores that have recently executed the same RPC type. We show this additional optimization speeds up the microservices' business logic by 1.05–2×, with the largest benefits experienced by the most common request types.

In summary, we make the following contributions:

- We study a suite of microservices and demonstrate that the RPC layer dominates their runtime, consuming between 40–90% of server CPU cycles. These cycles are spent in three key modules, two of which make up the vast majority of the cycles and can be handled by the same specialized hardware.
- We identify that the seemingly small third module, called "dispatch", must also be executed in hardware to avoid splitting the RPC layer's work between hardware and software. The dispatch module only accounts for <5% of the RPC layer's cycles but exposes offload overhead as a new bottleneck if left to run on the CPU.
- We design and evaluate Cerebros, an RPC processor that offloads the entire RPC layer to hardware and thus reduces the CPU cycles expended per request by 1.8–14×. Cerebros provides an additional 1.05–2× microservice processing time reduction by steering RPCs to cores based on instruction cache locality.

The rest of the paper is organized as follows: We first highlight the combination of need and opportunity that motivates hardware acceleration of the RPC layer (§2). We then introduce our proposed RPC processor design that can drastically improve handling of modern RPC layers as compared to a general-purpose CPU and prior hardware accelerators (§3). Building on our design principles, we then present our RPC processor implementation, Cerebros (§4). Next, we detail our methodology (§5) and evaluate Cerebros (§6). Finally, we discuss related work (§7) and conclude (§8).

2 WHY HARDWARE FOR RPCS?

Online services have been transforming from single-binary monoliths to a concert of fine-grained modules known as microservices. The microservices architecture simplifies development and deployment by creating independent modules responsible for subsets of the application's functionality and enforcing modularity [7, 24, 37]. Microservices are deployed across many servers and thus require inter-server communication using an API such as Remote Procedure Calls (RPCs). Decomposing a monolith into microservices implies that each microservice performs only a small fraction of the application-level work and that the total time spent on inter-microservice RPCs increases in proportion to the number of microservices. This increase in the communication-to-computation ratio creates a challenge to minimize the "tax" associated with each RPC.

Although the tax on inter-microservice communication includes both the RPC layer and the underlying network stack, ongoing research has drastically reduced networking latency. Modern datacenter network topologies [27, 67] and protocols for optimized congestion control [1, 31, 56] achieve network traversals of a few microseconds (μs) with high predictability. Furthermore, transport protocols in either user-space [17] or hardware [9, 29, 57] have drastically shrunk the cost of the transport layer from 10s of μs [64] to as low as sub-μs values [41]. Hence, the time spent in the RPC layer is becoming a significant fraction of the end-to-end cost of invoking a microservice. Prior work reports that microservices spend up to 75% of their on-CPU time in the RPC and transport layers [24]. In order to improve this emerging RPC bottleneck, the first priority is to decipher its underlying operations and identify their costs.

2.1 The Cost of RPCs

Figure 1a breaks down the layers of the system stack exercised by a microservice. Upon the arrival of a new request, the server terminates the transport protocol and hands the request to the RPC layer. After the RPC layer completes, the microservice's business logic executes and creates the response to be sent to the original

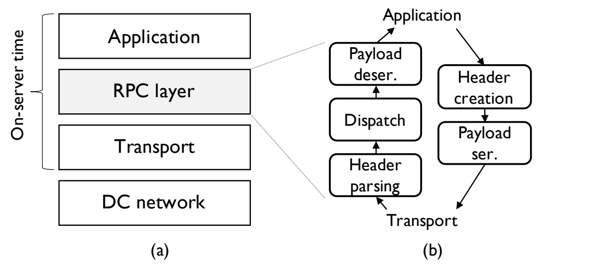


Figure 1: System stack exercised in a microservice's invocation and operations within the RPC layer.

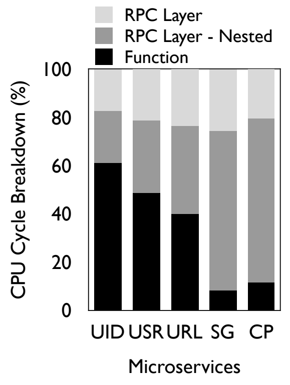
requester. The response goes through the same layers in reverse before the message leaves the server.

RPC frameworks such as Apache Thrift [4] or Google's gRPC [26] are themselves multi-layered architectures. Figure 1b expands the RPC layer to display the common functionalities comprising these frameworks. Upon receiving a new request, the RPC layer first parses the header, which contains fields indicating the message type and the requested function. Next is the dispatch module, which looks up the function ID in a table to retrieve the handler associated with this function and passes control to it. Finally, the handler prepares the input arguments by deserializing the message's payload and calls the requested function, terminating the RPC layer. Response messages are handled by the RPC layer in a similar manner, applying the same operations in inverse order. We categorize the aforementioned operations into three modules: header manipulation, payload manipulation, and dispatch.

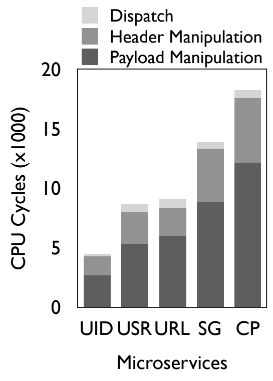
The above walkthrough presents a simplified case where the application is self-contained and completes its processing in isolation. However, microservices commonly perform several nested RPCs while processing a request for reasons such as retrieving data from other microservices. The inclusion of nested RPCs means the execution of the business logic is interrupted multiple times by repeatedly traversing the RPC layer. Such behavior further increases the time a microservice spends in the RPC layer. Oscillating between the microservice and the RPC layer also negatively impacts the CPU's instruction supply, leading to a higher number of instruction misses than would be experienced by an RPC-free application.

To quantify the RPC layer's cost, we study five microservices from DeathStarBench [24]: UniqueID (UID), User (USR), UrlShorten (URL), SocialGraph (SG), and ComposePost (CP). Methodology details are available in §5. Figure 2a breaks down each microservice's mean on-CPU time when processing various request types into the following three categories: (i) the RPC layer for the initial request message and its corresponding final response, (ii) the RPC layer for any nested RPCs that the microservice generates, (iii) the application-layer functions.

In all cases, the RPC layer takes a significant fraction of the microservice's runtime, accounting for 40–90% of the on-CPU time. The fraction of time spent in the RPC layer varies widely because the functions comprising these five microservices have different complexities, input/output message types, and number of nested RPCs. For example, UID has a single function that generates a globally unique integer and one nested RPC to upload that ID to another microservice. In contrast, CP has six simpler functions, but



(a) Function vs. RPC Layer.



(b) Inside the RPC Layer.

Figure 2: Breakdown of CPU cycles expended in microservices and the RPC layer.

each one has several nested RPCs to other microservices; hence, ~70% of CP's expended cycles are attributed to nested RPCs.

Next, we further classify RPC layer time into the cycles expended in each of the three aforementioned modules and display the results in Figure 2b. All cycle counts are cumulative over the request RPC, the final response RPC, and the nested RPCs that occur within the microservice's functions. Payload manipulation stands out as the largest component, accounting for ~60% of the RPC layer's total expended cycles. The absolute cost of payload manipulation is a function of each message's size and layout and adds up with each nested RPC. CP and SG's aggregate payload manipulation cycles in Figure 2b are greater than UID, USR, and URL because they create more nested RPCs, and each individual payload manipulation task is costlier due to larger and more complex messages. In contrast, header manipulation uses an identical format (i.e., data types and values) across all of the microservices, and therefore the total cost of header manipulation only depends on the number of nested RPCs. The same is true for the dispatch module. Therefore, microservices like SG and CP have a far greater aggregate cost for header manipulation and dispatch than those similar to UID.

This study shows that once microservices are deployed using optimized transport and network layers, the RPC layer is a prime contributor to a server's expended CPU cycles. Of equal importance is that nested RPCs cannot be overlooked: as microservices become more specialized and modular, the greater the cumulative RPC overheads. These overheads are concentrated in the RPC layer's payload and header manipulation modules, which together make up ~95% of RPC cycles. We conclude that improving the performance of microservices requires focusing on the RPC layer itself as a primary factor.

2.2 Toward Faster RPC Processing

Despite the critical nature of the RPC layer for microservice performance, CPUs are ill-suited for the three common RPC modules. Payload and header manipulation consist of *data transformation (DT)* tasks, and prior work has already demonstrated that CPUs perform such tasks orders of magnitude slower than necessary [60]. The dispatch module is also ill-suited for CPUs because it contains multiple data-dependent and indirect branch instructions that depend on the incoming message. In addition to these issues, it has already been reported that CPUs are plagued by instruction supply problems when executing microservices [24], which will worsen with the number of functions, message types, and nested RPCs that make up a microservice. When the inefficiencies of CPU-centric DT are combined with the instruction supply issues in microservices, using dedicated hardware for RPC tasks becomes an attractive solution.

To justify the investment in dedicated hardware, it must be widely applicable and also configurable for the sake of future software deployments. Our breakdown of the RPC layer shows these exact characteristics are true for its three modules; despite the diversity and rapid evolution of microservices, they all depend on these three ubiquitous modules. Furthermore, the maturity of the RPC layer [42, §4] indicates that dedicated hardware for its modules will not be immediately obsolete. Via the use of a dedicated abstraction to represent both payload and header manipulation [60], such hardware can be made applicable to any RPC message and framework. Hence, we argue it is feasible to design hardware that is drastically more effective in executing the RPC layer than CPUs.

Although the use of FPGA-equipped NICs has been proposed to accelerate RPC layer operations [20, 52, 62, 74], no existing design has managed to target all of the RPC layer modules we describe in §2.1. The most difficult challenge is to support the header and payload manipulation tasks because the message objects in production RPC layers are complex pointer-based data structures. Processing such software-readable objects requires judiciously co-designing RPC hardware and software around the constraints of the server's DMA engine, which only transfers opaque chunks of bytes or scatter-gather arrays [62]. Therefore, we argue that it is logical to handle such tasks with hardware integrated on chip, removing the DMA engine's constraints as well as the extra latency incurred whenever data must be moved across the I/O interconnect. Details of existing proposals are further discussed in §7.

Although it may appear logical to limit the scope of integrated hardware accelerators to the two manipulation modules because they make up ~95% of the execution time, the seemingly small dispatch module creates a critical bottleneck that must be addressed. To demonstrate this counterintuitive insight, Figure 3a displays the workflow of tasks occurring during RPC layer processing, assuming a state-of-the-art accelerator designed for DT [60]. Although originally designed only for payload manipulation, we assume this accelerator has the trivial extensions necessary to perform both header and payload manipulation, so both modules are accelerated (②, ④). However, because the dispatch module is logically wedged between the two manipulation tasks and remains on the CPU (③), the CPU serves as the coordinator that creates offload tasks after a new network message arrives (①). Using the accelerator as a co-processor in this manner inherently adds *offload overheads* to each manipulation task. These offload overheads are a critical obstacle preventing current accelerators from fully curtailing the cost of the RPC layer, particularly because their cost accumulates when a microservice uses many nested RPCs.

For a 16-core server with a mesh interconnect and an LLC-attached accelerator, we estimate that each offload incurs a cost of ~200 cycles (see §5 for details). When the manipulation tasks are small (e.g., for RPC headers), the offload overhead takes up to 90% of the entire module's execution time, bounding the benefit of acceleration to just 2× compared to the CPU. Therefore, offloading header manipulation, in addition to payload manipulation, provides limited performance gains due to the presence of the dispatch module between the two tasks. We conclude that although it is logical to invest in hardware for the two common manipulation tasks, keeping the dispatch module on the CPU cripples end-to-end performance due to cumulative offload overheads, and therefore it must also be done in hardware.

The inclusion of dedicated hardware for all three of the RPC layer's modules has the side benefit of improving the CPU's instruction supply. In all of the five microservices we studied, the RPC layer's instruction footprint forms a significant fraction of the microservice's working set. However, many individual functions are small enough to entirely fit inside the L1 instruction cache, which is not possible when the RPC layer's instructions are included. In a server with hardware support for the RPC layer, these instructions vanish, and any remaining L1 instruction cache contention occurs when the execution of multiple functions is interleaved on the same core. Inter-function contention can be solved by RPC hardware that is able to assign requests to cores in a manner that is aware of the requested function. Next, we present the design principles for hardware that accomplishes both the RPC layer and such function-aware request steering.

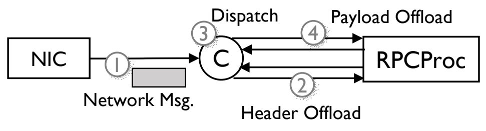
3 RPC PROCESSOR DESIGN

In this section, we present the design of a specialized RPC processor (RPCProc) to completely remove the RPC layer's burdensome tasks identified in §2. Our architecture is guided by the following four design goals: (G1) the CPU should only need to run the business logic of the microservice rather than the RPC layer, (G2) the RPCProc should be autonomous and not CPU-controlled, (G3) the RPCProc should be synergistic with state-of-the-art NIC architectures, (G4) the RPCProc should have minimal silicon requirements.

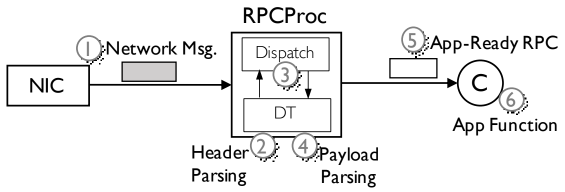
3.1 Eliminating Offload Overheads

In current systems, the NIC directly interacts with the CPU cores to signal the arrival of incoming work, as shown in Figure 3a. After terminating the network and transport protocols, the NIC sends a request arrival notification to a CPU core, and then the core begins processing the RPC layer. Consequently, using the RPCProc for RPC layer acceleration mandates at least one explicit task offload from the CPU to the RPCProc. Following our analysis in §2.2, a critical requirement for the RPCProc is to receive incoming requests directly from the NIC and process the full RPC layer to completion before involving the CPU. The same is true for outgoing requests, except the RPC layer must entirely complete with a single RPCProc call by the CPU to start the sending process.

Realizing the goal of running the entire RPC layer in hardware requires the RPCProc to be a transport protocol endpoint. The use of lean hardware-terminated protocols enables this design change because there is no transport processing remaining once the incoming message exits the NIC. Any well-established signaling method (e.g., in-memory queues [19] or MSI-X interrupts [11]) is sufficient to interface the NIC and RPCProc, thus meeting G2 and G3. However, an RPCProc that directly receives incoming requests from the NIC is still insufficient to achieve G1. The RPCProc must also



(a) Explicit CPU-controlled offloads.



(b) NIC-interfaced.

Figure 3: Comparison of CPU-controlled versus NIC-interfaced accelerators for incoming RPC.

support the RPC layer's dispatch module in the same location as the manipulations—otherwise, the system reverts to the one in Figure 3a, where the dispatch and manipulations are logically split, necessitating the RPCProc to be under CPU control.

Figure 3b displays the architecture of an RPCProc that achieves G1 and G2: it receives incoming RPC requests directly from the NIC (①) and processes the RPC layer without the CPU's involvement, starting with header parsing (②). It then performs the dispatch module using hardware, which reads the function ID from the parsed header and looks it up in a dispatch table to find the metadata describing how to parse the corresponding payload type (③). Using this information, the accelerator parses the payload (④) and passes an application-readable RPC to the CPU (⑤) that executes the requested function (⑥).

An RPCProc with a direct NIC interface not only eliminates offload overhead, but also benefits the microservice's on-CPU business logic execution by improving the CPU's instruction supply behavior. As mentioned in §2.2, performing the RPC layer in hardware completely bypasses the set of instructions dedicated to RPC processing. Additionally, as the RPCProc already has knowledge of the requested function ID from header parsing (②), it can choose the core to process this function based on any policy—in particular, we identify temporal locality as a beneficial one. Assigning RPCs to cores that have just executed the same function virtually guarantees that the core's instruction cache is warm and that function will execute with fewer CPU frontend stalls. We call this approach *affinity-based request steering*. Next, we present our RPCProc's components and system integration, which are critical to meet G3 and G4.

3.2 Components for RPC Tasks

An RPCProc's most important component is the module that handles payload and header manipulation because those two tasks constitute the vast majority of RPC latency. Both manipulation tasks essentially reduce to the same low-level operation of converting an in-memory object to its wire format, or vice versa. Due to the prevalence of such manipulation tasks and their associated CPU limitations, bespoke accelerators for payload manipulation or object (de)serialization have been proposed in prior work [36, 60]. Although such designs provide impressive speedups for manipulation tasks, neither operates autonomously, meaning they do not satisfy G1 and need to be expanded to be included in a full RPCProc.

Executing the full RPC layer in hardware requires control logic surrounding the RPCProc's manipulation hardware that performs two tasks previously left to the CPU: initiate accelerator processing in response to incoming requests, and perform the RPC layer's dispatch module. To initiate a new manipulation task, the control logic must communicate the input/output buffers and object to be transformed to the manipulation hardware. State-of-the-art accelerators already use in-memory metadata (called *schemata*) [36, 60] to represent the data to be manipulated, and therefore commencing a new manipulation task simply requires indicating the correct schema and buffer addresses to the accelerator. As the schemata are flexible enough to represent both header and payload manipulation, the same initiation logic is sufficient to create tasks for 95% of the work in the RPC layer.

Moving the dispatch module to the RPCProc's hardware is necessary to eliminate offload overheads and meet G2. In software, the dispatch module reads the parsed header to retrieve the function ID, calls the subroutine that deserializes the message's payload, and then transfers control to the function handler. To realize this in hardware, the in-memory schema must be extended to include a marker specifying which header field identifies the function. After the header-parsing task, the RPCProc's dispatch logic extracts the function ID and matches it with the corresponding schema describing this function's payload manipulation task. For this purpose, the RPCProc contains a small table that is looked up by function ID and returns the correct schema and address of the respective function handler.

3.3 Server System Integration

Figure 4 shows the architecture of an on-chip RPCProc and the components that execute the header manipulation, dispatch, and payload manipulation modules of the RPC layer. Our RPCProc design is architected around an on-chip integrated NIC with a hardware-terminated transport protocol (G3) because such designs are a natural starting point for servers optimized for microservices. Architectures featuring integrated NICs are already commonplace, with academic examples like the FAME-1 RISC-V RocketChip SoC [43], Scale-Out NUMA [57], and the NanoPU [33]. Commercial examples include Oracle's Sonoma [30], Calxeda's ARM SoC [73], and Intel Xeon-D servers with integrated Ethernet [35].

We integrate the RPCProc with the on-chip NIC to reduce silicon costs and deployment complexity (G3 and G4) because the RPCProc and NIC share glue logic that connects them to the CPU's memory hierarchy. In particular, both components need a small cache and its matching MMU, which the NIC uses to read/write data coherently and the RPCProc will use to operate on that data when it performs the RPC layer. By moving the endpoint of the transport protocol to the RPCProc, it now must be the agent which communicates

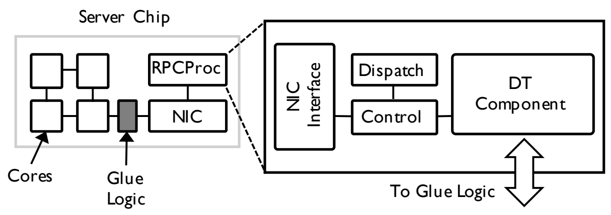


Figure 4: Architecture of an on-chip RPC processor.

with the CPU cores to inform them of incoming RPCs (c.f., step ⑤ of Figure 3b). As CPU-NIC communication has been shown to be problematic for small data transfers, it is logical for an RPCProc meeting G3 to leverage highly optimized architectural support in state-of-the-art NIC designs [13, 71].

An alternate design point is to provision an RPCProc per CPU core, sharing the CPU's glue logic rather than the NIC's. However, we choose to use a single shared RPCProc for two reasons. First, the silicon overheads of a design with replicated RPCProcs are considerable, thus contradicting G4. Prior work only considering payload manipulation has shown that with a 64-core server, the total area cost of such replicated accelerators corresponds to 75% of an entire server-grade CPU core [60]. The overheads would naturally be greater for a full RPCProc, which requires additional logic for control and the dispatch stage. Second, a per-core design precludes the RPCProcs from employing affinity-based request steering because requests are first sent to the per-core RPCProcs by the NIC before the function IDs are known.

Integrating the RPCProc with the server's NIC enables function-to-core affinity as one of potentially many policies in the stages of the NIC that assign incoming requests to cores. State-of-the-art NICs already contain support for assigning work to cores based on metrics like load balancing [14] or TCP connection locality [34], and therefore it is logical to provide affinity-based request steering in the same location. The RPCProc components in §3.2 already extract the function ID from the header parsing stage to execute the dispatch module and can provide it to the NIC's core-assignment stage once RPC layer processing is concluded.

4 THE CEREBROS RPC PROCESSOR

In this section, we present Cerebros, an implementation of a full RPC processor following §3's design principles. We first briefly introduce the critical features of our assumed network hardware and discuss Cerebros' interface with the NIC and the CPU cores (§4.1). We then describe Cerebros' components that replace the RPC layer's modules (§4.2 and §4.3) and conclude with the extensions for affinity-based request steering. Figure 5a presents the Cerebros architecture, with indicators showing the process of receiving and processing an RPC. Alphabetic indicators show events associated with the NIC, whereas numeric indicators show Cerebros' operations.

4.1 NIC and Software Interfaces

As motivated in §3.3, it is logical for Cerebros to be constructed over a baseline system featuring an on-chip integrated NIC and hardware-terminated protocol. We, therefore, select the NeBuLa [71] architecture as our baseline because it features an RPC-oriented hardware-terminated transport and an integrated NIC attached to the server's on-chip network. Software endpoints communicate with the NeBuLa stack by using an RDMA-like memory-mapped Queue-Pair (QP) interface [19].

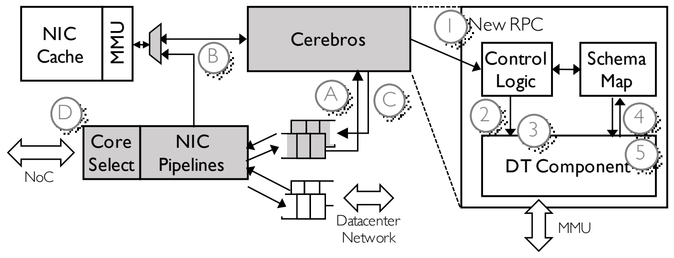
When packets arrive at the server, NeBuLa's NIC pipelines terminate the transport protocol, reassemble the possibly fragmented network packets into a full message, and place it into a dedicated NIC cache that is coherent with the server's memory hierarchy. NeBuLa keeps newly arrived messages in a NIC-private memory-mapped queue until a core becomes available to process a new message. When a core indicates its availability, NeBuLa creates a new entry in that core's QP, pointing to the received message's buffer location in memory. The core polls its QP to receive the RPC arrival notification.

To meet the goal of performing the RPC layer without CPU involvement (§3, G2), Cerebros needs to be inserted into the flow of incoming RPCs as a logical step between NeBuLa's transport protocol termination and core notification. As our design goals are best fulfilled by integrating the RPC processor with the NIC, we choose to add a simple interface comprising two hardware queues between NeBuLa's NIC pipelines and Cerebros' control logic. Cerebros only begins RPC processing *after* network protocol handling completes; the inverse is true for outgoing RPCs.

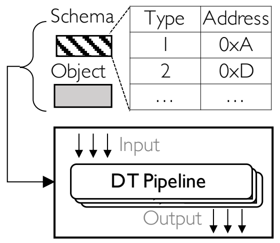
As in the NeBuLa baseline, the NIC pipelines place incoming RPC messages into the NIC cache. The NIC invokes Cerebros' control logic through a hardware queue (Figure 5a, Ⓐ), passing the address of the newly arrived message. Cerebros' data accesses all go through NeBuLa's existing MMU (Ⓑ) and find the target data already resident in the NIC cache. Once Cerebros completes its processing tasks, its control logic returns a message to the NIC pipelines indicating RPC processing is complete, which contains a metadata structure with all of the RPC's corresponding data (Ⓒ). The NIC pipelines' final stages then execute the core selection logic and use NeBuLa's default mechanism to notify the selected core through its QP (Ⓓ). §4.3 details how we adapt NeBuLa's core selection logic to accommodate affinity-based request steering.

Software Interface. Cerebros' control path is used at initialization time by microservices that wish to offload their RPC layer. Software must provide Cerebros with the following information: i) its function IDs and their respective payload types, ii) the metadata (schema) describing each function's payload layout, iii) the globally shared format for header manipulation, and iv) a set of memory arenas used by the manipulation accelerator to place its output into. Each of these parameters is created once on application start and programmed into Cerebros' memory-mapped control registers via `ioctl` system calls.

Each of the microservice's threads creates and registers a dedicated QP that is used for sending and receiving network messages. Incoming messages placed in the thread's QP have been completely processed by Cerebros and can be directly processed by the function whose ID is indicated in the new QP entry. Outgoing nested RPCs and responses are placed by the microservice directly in the QP without invoking software RPC processing, which is completely performed by Cerebros before the message is delivered to the NIC for transport encapsulation. In case Cerebros cannot process a message (e.g., due to an unrecognized function), a fallback mechanism



(a) Cerebros. Shaded components are modified or newly added.



(b) DT Component.

Figure 5: Architecture of Cerebros, and its component for header/payload manipulation.

sends the unprocessed message to a thread, indicating with a null function ID that the RPC layer must be executed in software.

Memory Management. During its payload manipulation stage, Cerebros unpacks the incoming message's arguments and prepares them for the software to read. However, this implies that a buffer must be provisioned for the deserialized payload, in addition to the transport buffers reserved for and managed by NeBuLa's network protocol stack. Instead of adding another disparate memory reservation stage in Cerebros' hardware, a more efficient alternative is to unify this buffer management with NeBuLa's transport buffer management and make them "all-or-nothing" atomic. Allocating both the transport and application buffers together avoids the need for additional logic in the NIC to handle cases where transport allocation succeeds but RPC layer allocation fails, which is likely to be rare and complex to handle. To unify the two buffering stages, we extend NeBuLa's buffer manager (which originally only manages transport buffers) to also reserve memory for the deserialized payload. If either memory reservation fails, NeBuLa returns a NACK to the sender according to its existing protocol; the sender reacts to the NACK according to a policy of its choice.

To ensure the application buffer's allocated size is sufficient to contain the deserialized payload, we use the insight that in production RPC stacks, the maximum possible field-level compression is 4×. This compression occurs only in variable-length integers, which can shrink from 8B in their application format to 2B in the network format. All other primitive types have lower compression factors due to additional metadata, and the same is true for composite types such as Maps. Therefore, Cerebros allocates 4× the network message's size for the deserialized payload, which is guaranteed to be sufficient memory even if the entire incoming message consists of variable-length integers. All RPC layer memory comes from the arenas pre-allocated and installed by the microservice through Cerebros' control interface. Next, we discuss the architecture of the components comprising Cerebros.

4.2 Data Transformation Component

Due to the commonality between header and payload manipulation, the two operations can be handled by a single hardware component performing data transformations [60]. The two most relevant components in the literature that specifically target data transformations with bespoke hardware components are Optimus Prime [60] and Cereal [36]. We hereafter refer to a component of this form as a "data transformation accelerator" (DTA). While both DTAs address the same problem and arrive at similar hardware designs, Cereal only works with a dedicated serialization format, limiting its generality. A DTA following Optimus Prime's design patterns is more applicable to datacenter microservices because it does not require changing each microservice's data format to match the specific DTA implementation. Therefore, Cerebros adopts a DTA design similar to Optimus Prime for header and payload manipulation.

The DTA's key enabling feature is the use of a transformation schema, an in-memory data structure that represents the parallel sub-tasks comprising each manipulation request. Figure 5b shows a sample manipulation task that serializes an `Object` from language-readable to wire format. The RPC framework creates a schema for each instance of an `Object`, defining the list of tasks required to serialize it. Each row of the schema represents one of the `Object`'s fields, indicating its `Type` and the `Address` where the data is to be read from. Cerebros borrows this transformation schema as a flexible accelerator interface that allows defining all types of parallel data manipulation tasks, facilitating compatibility with any RPC framework after the schema's format is established.

Internally, the DTA is organized as an array of independent transformation pipelines, each featuring a set of hardware units operating in a decoupled access-execute mode [68]. During serialization, its input units read data from the memory hierarchy based on the *Address* fields and feed transformation units, which feature simple ALUs that transform data according to the installed schema's rules. The output units write the transformed data to the designated memory buffer. An architecture like the DTA we have presented is sufficient to handle both RPC manipulation modules; the next component that must be addressed is the one handling dispatch.

4.3 RPC Dispatch and Request Steering

Moving the dispatch module into hardware is mandatory for complete RPC layer processing on Cerebros without CPU involvement. We now walk through the tasks performed by Cerebros when the dispatch stage executes, using Figure 5a as a guideline. When a new RPC task arrives at Cerebros from the NIC (①), Cerebros' controller assigns the RPC to an available transformation pipeline and passes the request's metadata to it (②). After the DTA parses the header (③), Cerebros must (i) determine the function ID being requested, and (ii) prepare the payload manipulation task corresponding to that function's message type.

To meet these two requirements, we extend Figure 5b's schema format to include a special marker indicating which field of the header contains the function ID. Our added dispatch logic in the DTA pipeline uses the modified schema to extract the function ID from the deserialized header. After the function ID is known, Cerebros uses a small table, called the *Schema Map*, that maps this ID to the correct schema corresponding to the incoming request's payload format. The Schema Map is exposed via an in-memory configuration space and programmed by the microservice at start-up through Cerebros' control path (§4.1). The Schema Map's storage requirements are limited because we expect the number of concurrently active functions to be a few tens.

We also introduce the idea of a *split schema*, which decomposes each schema into two parts. The first part is the `Type` column of Figure 5b, which only represents the data types pertaining to a particular message class. As all of the messages reaching a particular function are of the same type, the `Type` schema remains immutable and is shared among all messages of the same type, including headers. The second part is unique for each individual message and contains the data pointers (the `Address` column). Dividing schemata in this fashion roughly halves their storage requirements, as Cerebros will access the same read-only `Type` schema for all incoming messages to the same function. Additionally, such division eliminates the need for Cerebros' DT component to create a new `Type` schema in memory for every request.

Returning to Figure 5a, after header parsing completes, Cerebros uses the special marker in the header schema to extract the function ID. It then looks up the Schema Map (④) which returns the `Type` schema for the corresponding function's payload type. To prepare the payload manipulation task for the incoming RPC, Cerebros creates a blank `Address` schema in the memory previously reserved by the NIC's pipelines for the DTA to fill out with each `Type`'s address. Cerebros' control logic initiates payload manipulation by passing pointers to both the raw payload and its application-level memory containing the `Address` schema and intended output buffer to the DTA. The DTA then parses the payload and fills out the `Address` schema with the addresses where the application-readable fields were placed. When payload manipulation completes, Cerebros sends two pieces of information to the NIC's core selection stage (Ⓒ): a pointer to the buffer with the application-readable request and the function ID.

Affinity-Based Request Steering. The final task remaining is to select a core to send this RPC to—the result of this process is what allows us to realize affinity-based request steering. NICs already implement logic to perform core selection based on a variety of metrics (e.g., load balancing [14] or TCP 5-tuple [34]). Cerebros contains a core selection stage that obtains a set of desirable cores for handling this function from a table called the *function map* (Ⓓ).

The function map is a direct-mapped table storing a FIFO list of recently executed function IDs for each CPU core. When a new RPC is assigned to a core, the function ID is added to the head of the core's list, and the tail of the list is dropped. Our implementation only stores a single entry per core, so that a core is only considered as having affinity if it has just executed the exact same function.

Selecting a core for a new RPC involves comparing the function map's entries against the incoming function's ID, and considering that a core has affinity to this function if the incoming ID matches. To preserve load balancing, Cerebros' core selection stage then chooses the core with the fewest number of outstanding RPCs from the set of all cores having affinity to this function. Such policies that assign requests based on the number of outstanding requests per core have been implemented in hardware by prior work [14, 48]. Further core assignment policy optimizations (e.g., increasing the depth of the list in the function map in the case where multiple functions have constructive code sharing) are interesting extensions to our proof-of-concept implementation.

While the core is being selected, NeBuLa's NIC pipelines create a metadata structure containing a pointer to the incoming message's *Address* schema, the corresponding request buffer, and a function pointer that indicates the address where the core must begin executing. Cerebros notifies the selected core of a new incoming request, passing the metadata to it via a QP entry. Once the core receives the notification, it begins executing the function indicated in the metadata structure.

5 METHODOLOGY

Evaluated Microservices. We choose microservices from DeathStarBench [24] that differ in the following primary parameters that dictate the RPC layer's cost breakdown (c.f., §2): number and complexity of functions, frequency of nested RPCs, and message size/format complexity. Our microservices are UniqueId (UID), User (USR), UrlShorten (URL), SocialGraph (SG), and ComposePost (CP), which comprise one, six, one, seven, and six underlying functions, respectively. The selected microservices represent DeathStarBench's various microservice classes. Other microservices in this benchmark suite behave identically or similarly to those we evaluated. In particular, most of the microservices are similar to SG and CP, which contain little business logic and spend most of their execution time just passing data along to other microservices or data stores via nested RPCs. Facebook has also recently revealed their web services (the closest workload to DeathStarBench's microservices) spend as little as 18% of their execution time in the application logic [69]. All microservices use Apache Thrift [4] as their RPC layer, to which we have added a new hardware-terminated transport protocol based on NeBuLa [71]. We study each microservice in isolation and create mock components for the other microservices surrounding the isolated one. Due to our use of isolated microservices, we report the CPU cycles expended in only the RPC and application layers. Therefore, our results are independent from the underlying transport and network protocols.

Request Processing Model. Our evaluated RPC layer implements a synchronous request processing model, where each microservice polls for incoming requests and executes them to completion. Threads also synchronously poll for the results of their nested RPCs, which Cerebros guarantees will be returned to the same thread. An asynchronous processing model (where threads begin processing new requests instead of polling for responses to nested RPCs) would provide higher throughput at the cost of extra CPU cycles spent for context switching and higher programming complexity [8]. A user-level threading library such as Arachne [61] would be mandatory for handling the μs-scale execution times of our evaluated microservices. We emphasize that because Cerebros's primary target is the reduction of CPU cycles expended per request, it benefits

Table 1: Parameters used for cycle-accurate simulation.

Cores	ARMv8, 64-bit, 2GHz, 4-way OoO, 128-entry ROB TSO, next-line instruction prefetcher
L1 Caches	64KB 4-way L1-D, 64KB 4-way L1-I, 64B blocks 2 ports, 32 MSHRs, 4-cycle latency (tag+data)
LLC	Shared block-interleaved NUCA, 8MB total 16-way, 1 bank/tile, 8-cycle latency
Coherence	Directory-based Non-Inclusive MESI
Memory	45ns latency, 2×25.6GBps DDR4-3200
Interconnect	2D mesh, 16B links, 3 cycles/hop

either processing model, and the saved cycles can be re-purposed to increase concurrency if asynchronous RPCs are used.

Microservice Characterization. To accurately measure the breakdown between the functions and RPC layer, we instrument the microservices' code to record cycles expended in the following three steps: (i) the RPC processing that occurs upon new requests arriving, (ii) nested RPCs that occur during the function's execution, and (iii) the function code itself. Therefore, the cycles we attribute to the function quantify only the time spent executing the application's business logic. Reported cycle counts are the average number of cycles expended per request across all functions for each microservice. To estimate instruction working set sizes, we apply the methodology used for profiling workloads in Google datacenters [42]: we collect the trace of executed instructions and measure how many unique cache lines cover 99.9% of the trace when ranked by popularity.

Simulation Setup. We evaluate Cerebros using cycle-accurate full-system simulation. We use the QFlex simulator [58] to simulate a 16-core ARMv8 CPU running Ubuntu Linux 18.04, whose parameters are summarized in Table 1. All workloads are pinned on 15 cores, leaving one core for system tasks and interrupt processing. We limit UID to four cores because lock contention limits its scalability. Our simulator includes a load generator that creates incoming requests based on a given popularity distribution, dictated by the structure of the microservice [59, §6.1], and delivers notifications to the CPU through the NeBuLa transport stack. The load generator also emulates all the mock microservices, mimicking their behavior and instantly responding to RPCs with pre-constructed messages.

To compare against Optimus Prime (OP) [60], we estimate its best-case performance using numbers available in the respective paper. We model OP's processing time as the cycles required by its transformation pipelines to process all the fields of the message, plus a single access to the cache hierarchy for all required data. We use a fixed latency for each message transformation that varies based on the message type and is calculated based on the message's structure and fields, following Thrift's compact protocol encoding. For Cerebros, we add the cycles required for header parsing and dispatch, calculated identically. We assume no queuing delays in either OP or Cerebros due to the fact that the accelerators' message processing rates are 2.5 – 100× faster than the maximum load generated by the cores running our microservices. Moreover, in a real deployment, Cerebros would experience far less load because of the extra latency contributed by the other surrounding microservices and the datacenter network (as opposed to responses arriving instantly). Hence, Cerebros' processing rate dwarfs the cores' peak RPC generation rate, making queuing negligible.

Analytical Model. To explicitly study the impact of offload overhead, we use an analytical model similar to Accelerometer [69] to estimate the total expended cycles in the RPC layer, as particular layers of the RPC layer are offloaded to hardware. The message processing cycle counts are calculated using the same performance model previously explained for OP. To estimate the cost of a single synchronous offload, we model five sequential traversals of the server's on-chip network: (i) the CPU invokes the accelerator through MMIO writes; (ii) the accelerator reads the metadata describing the task, which is delivered separately from the invocation [60, §4]; (iii) the accelerator reads the data block(s) corresponding to the task; (iv) the data to be returned is written back to the cache hierarchy; and (v) the accelerator notifies the CPU. Each of these traversals incurs a latency of 40 cycles, measured using our cycle-accurate simulator.

6 EVALUATION

We begin our evaluation by quantifying the performance implications of offload overheads, demonstrating the need for Cerebros to directly interact with the NIC and run the entire RPC layer. We then show Cerebros' ability to achieve our first design goal: the CPUs only run the microservices' business logic and not the RPC layer. Next, we demonstrate how fully offloading the RPC layer actually improves the performance of the microservices themselves, and conclude by evaluating affinity-based request steering.

6.1 Impacts of Offload Overhead

The performance improvements from a design that uses the CPU as a coordinator for an RPC accelerator depend on the accelerator's per-module speedup and the overhead of each module offload. In Figure 6 we instantiate our analytical model for the following five designs executing the UID microservice: the CPU baseline (None), offloading the RPC's payload to the OP accelerator (OP: P-Only), offloading both the payload and header (P+H), using a private accelerator per core (P+H_PV), and Cerebros that performs the entire RPC layer (Full).

Accelerating payload manipulation (P-Only) leaves the rest of the RPC layer processing to the CPU and only reduces RPC layer cycles by 1.7×. Additionally offloading header manipulation (P+H) frees up 40% of the remaining cycles, bringing the total speedup to 2.3×. In this case, the only remaining part that is executed on the CPU is the dispatch module, which takes only 5% of the RPC layer's cycles. However, the offload overhead grows because the CPU must explicitly request the processing of both manipulation modules independently, forming a lower bound on the performance of the RPC layer. Each request to the UID microservice generates four header manipulation and four payload manipulation tasks, resulting in a total offload overhead of ~1600 cycles, compared to an accelerator processing time of only 90 cycles. For microservices with more nested RPCs (e.g., SG or CP), offload overheads dominate the cost even more overwhelmingly.

A brute-force solution to mitigate offload overheads is to integrate a private accelerator with each CPU core. Figure 6's P+H_PV bar shows the performance of this solution, where the only module

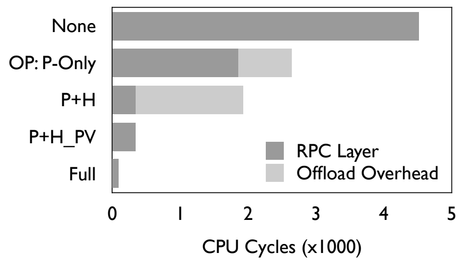


Figure 6: RPC layer cycles for various offload options, when executing the UID microservice.

remaining that contributes to RPC layer cycles is the dispatch layer. However, such accelerator replication requires 5× more area and 6× more power than a single shared accelerator with a 16-core chip, which is a steep cost for hardware operating at only 1% utilization in the case of UID. These costs grow with higher core counts, making replication an inefficient design choice.

We expect that the best solution to reduce offload overhead without a direct NIC-accelerator interface requires optimizing the accelerator's interface to coalesce its input data with the CPU's requests (similar to RDMA NICs) [39], and using a low-diameter on-chip interconnect such as Multi-drop Express Channels [28] or NOC-Out [54]. The combination of these two techniques can reduce offload overheads by 3.36×, reducing the exposed offload overhead to 60 cycles per module. Therefore, the performance of such an optimized system would fall between P+H and P+H_PV.

Only the solution with a fully capable RPC processor (Full) can simultaneously offload all elements of the RPC layer and avoid silicon overprovisioning. A NIC-interfaced RPC processor cuts expended cycles in the RPC layer by 50× when compared to the CPU baseline, and 21.5× compared to the case where only dispatch is performed on the CPU (P+H). Integrating the RPC processor with the NIC itself allows a 50% reduction in the area of the DT component [60], as it now shares the NIC's cache and MMU (§3.2).

6.2 Reduction in RPC Processing Time

We now evaluate the impact of Cerebros' ability to execute the RPC layer in hardware on overall microservice behavior. Figure 7 shows the mean on-server expended CPU cycles per request, broken down into the RPC layer and the application-level function. Cerebros virtually eliminates the cycles spent in the RPC layer and thus reduces the expended CPU cycles by 1.8–14.2×, depending on the fraction of cycles attributed to the RPC layer in the baseline.

The effect of RPC layer offload to Cerebros is most pronounced for SG and CP, as they spend ~90% of their cycles in the RPC layer due to heavy use of nested RPCs. The functions of these two microservices primarily pass along the information contained in their input messages to other microservices, performing tiny amounts of business logic. A function in CP can include up to 13 nested RPCs, accounting for up to ~70% of the microservice's total expended cycles on average, as shown in Figure 2a. In contrast, SG has a maximum of five nested RPCs, but the messages it exchanges with other microservices include complex nested objects and are larger than CP's. Message size and complexity make SG's breakdown of RPC versus function time similar to CP's, despite fewer nested

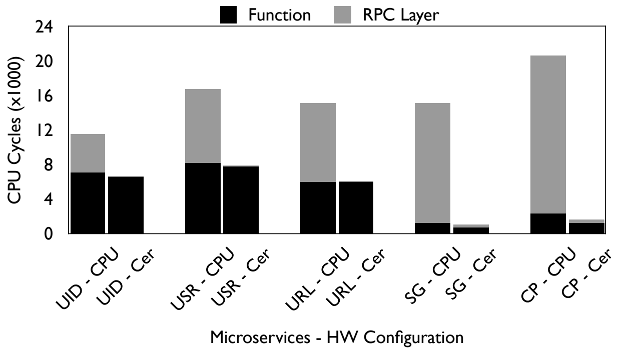


Figure 7: Average on-server cycles per request.

RPCs. Cerebros is able to effectively eliminate the RPC layer's overheads, whether the underlying root cause is deep RPC nesting or transformation complexity.

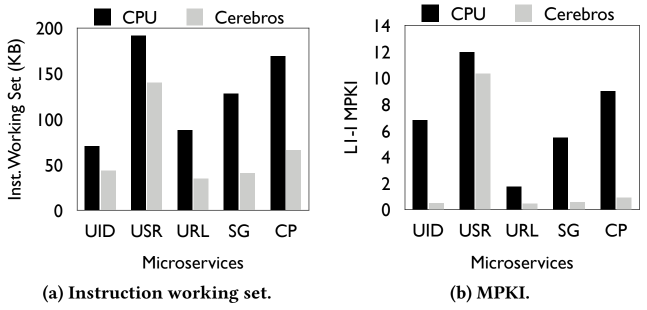
Business logic in UID, USR, and URL is more complex, hence forming a more notable fraction of the expended cycles. Additionally, UID's messages do not contain complex objects and are smaller than 50B in size. Despite the relative simplicity of UID's RPC tasks, Cerebros still attains a 1.8× reduction in CPU cycles.

As a side-effect of full RPC layer offload, Cerebros reduces the on-CPU service time of each microservice's business logic as well. Figure 7 shows 2–49% fewer expended cycles in the functions as a result of improved CPU frontend performance due to reduced instruction working set. To clearly show the effect on the CPU frontend, we measure the working set sizes and the MPKI values of our five microservices in two configurations: when the RPC layer is performed by the CPU and when it is offloaded to Cerebros.

Figure 8a shows the instruction working sets of our evaluated microservices. In the baseline CPU system, the bloated RPC layer results in total working sets that exceed the L1-I's capacity by up to 3×. In contrast, Cerebros' RPC layer offload reduces the working set by 27–68%, which naturally translates to a higher L1-I hit rate. The working sets are most visibly reduced for SG and CP, as they have little business logic in their functions and their instruction footprints correspond more directly with RPC layer code, due to their large number of nested RPCs and complex message types. Hence, when the RPC layer is offloaded to Cerebros, we see a reduction of more than 60% in their instruction working sets. On the contrary, UID includes only one nested RPC and uses simpler messages, while the function itself is roughly 43KB in size. Even then, offloading UID's RPC layer to Cerebros shrinks the instruction working set by 38%.

Figure 8b depicts the L1-I MPKI before and after the RPC layer offload. The working set reduction achieved by Cerebros directly affects the core's frontend performance, virtually eliminating instruction misses for four of the microservices and reducing CPU cycles wasted on instruction misses by 5–93%. Instruction miss reduction also yields a 2–49% reduction in function cycles, as shown in Figure 7, highlighting that RPC layer offload has a significant positive side-effect on the CPU performance.

USR benefits the least among all microservices because it includes two functions with working sets larger than 90KB in size. It also experiences the smallest reduction in the instruction working



(a) Instruction working set. (b) MPKI.

Figure 8: Frontend behavior of microservices.

set, as shown in Figure 8a. In such cases where the aggregate working set of all the functions still outstrips the L1-I cache, even fully offloading the RPC layer to Cerebros provides limited benefits to the CPU's frontend. We now evaluate the performance of affinity-based request steering that ameliorates CPU frontend inefficiencies in these exact cases.

6.3 Affinity-based Request Steering

Although the aggregate working set of the USR microservice when using Cerebros is ~140KB, four of its six functions are small enough to fully reside in a 64KB instruction cache if running in isolation. However, the working sets of the other two functions are >90KB. When the NIC's core selection policy does not take into account function locality, all six functions will compete for L1-I cache capacity, resulting in the high number of instruction misses visible in Figure 8b even after Cerebros' RPC layer offload. This phenomenon particularly hurts the performance of functions for which L1-I misses account for a large fraction of total execution time.

Figure 9 breaks down the CPU time of USR's six functions into execution time and time stalled on instruction misses and compares a Cerebros baseline (C) against Cerebros with affinity-based steering (CA). In the affinity-agnostic baseline, USR's functions are stalled on instruction misses for 15–44% of their total runtime. Function 2 is the only strongly compute-bound function, spending the majority of its time hashing strings after its working set is first loaded into the L1-I. All other functions have their CPU times divided roughly equally between execution and instruction stalls.

With affinity-based request steering enabled, the fraction of time stalled on L1-I misses drops by 1.05 − 18×, with the larger benefits being applicable to the most commonly executed functions, F3–F5. We measured that ~98% of requests were able to be steered to a core that had just executed the same function type, highlighting the fact that function affinity is plentiful for our deployment. For F3–F5, affinity-based steering virtually eliminates L1-I misses, leading to a 1.8 − 2× reduction in CPU time. These functions benefit drastically because their instruction working sets are between 20–25KB, which are easily accommodated by our CPU's 64KB L1-I cache. Affinity-based steering allows F3–F5 to execute with zero L1-I misses for 94% of requests.

Despite their high number of L1-I misses in the baseline, F0–F1 benefit only marginally from affinity-based steering because their L1-I misses primarily come from limited cache capacity, not inter-function contention. We have verified this with an experiment

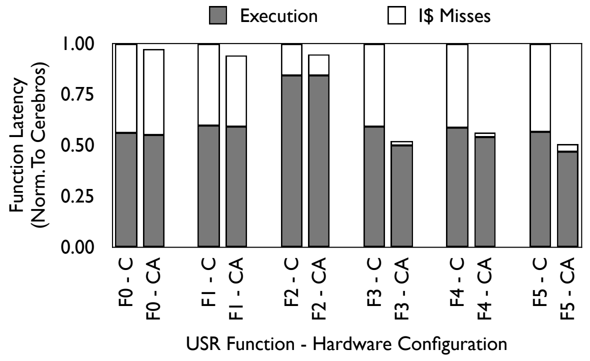


Figure 9: Breakdown of the USR microservice's functions into execution time and instruction cache misses.

enforcing that these two functions execute on dedicated cores to eliminate any contention from other functions. Even in this best-case scenario, F0–F1's CPU times are within 3% of what we observe with affinity-based request steering.

Aggregated across all of the functions, affinity-based request steering reduces average CPU time for the USR microservice by 8.7%. The fact that USR's two largest functions (F0–F1) have execution times ~180× larger than its most popular functions (F3–F5) skews the average downwards. In contrast, the median CPU time drops by 33% because F3–F5 comprise 70% of total incoming requests and experience greater speedups.

7 RELATED WORK

Other RPC Acceleration Frameworks. The prevalence of RPC-connected microservices in the datacenter has led to a plethora of proposals to use production NICs, equipped with FPGAs or their own CPU cores, to accelerate application-level tasks such as the RPC layer. Dagger [52] is the only other work we are aware of which proposes to offload the full RPC layer to hardware. They target the integrated FPGA in an Intel Broadwell platform and build a customized RPC layer inspired by the Thrift [4] software stack. Similarly, NICA proposes a programming model and runtime to accelerate application-level tasks on FPGA-enhanced NICs [20], citing message de-serialization as a potential application. Neither Dagger nor NICA currently supports the underlying modules of production RPC layers we describe in §2.1 because they lack support for the header and payload manipulation modules in production Thrift [52, §4.5]. Dagger and NICA share our use of hardware-terminated transport protocols, but still leave header/payload manipulation (and therefore RPC dispatch) to the host CPU.

Adding support for manipulation tasks to designs based on commodity NICs would require co-design with the host's DMA engine, because common-case objects cannot be deserialized on the NIC and then DMA'ed to the host without rewriting all of the object's pointers. Although it may be possible to realize such a design in the future, as argued by Wolnikowski et al. [74] and Raghavan et al. [62], the hardware required will likely be similar to Cerebros, and therefore we believe an on-chip design is more logical. Additionally, executing the RPC stack on commodity NICs will inevitably incur the overhead of transferring objects between the FPGA accelerator and the host's CPU cores. Dagger and NICA contain a highly customized communication stack to transfer objects between the FPGA accelerator and the host's CPU cores, and reduce the latency of the FPGA-to-host interconnect—another form of offload overhead that Cerebros does not incur due to its use of an on-chip NIC.

Customized RPC frameworks such as Cap'n Proto [46] or FlatBuffers [25] use a pre-flattened RPC message format, which inherently removes the aforementioned challenges with pointer-based objects. However, such frameworks sacrifice object mutability, generate larger wire-format objects, and experience higher latencies during costly object creation [62]. We believe these trade-offs are particularly burdensome for microservices with many nested RPCs because message objects are commonly modified on every nested call. Cerebros can even benefit such frameworks by accelerating header parsing and with affinity-based request steering.

Optimus Prime [60] introduces a data transformation accelerator (DTA) to perform RPC payload manipulation at rates matching a high-end server NIC. Cerebros builds on Optimus Prime's DTA architecture and transformation schema to perform both payload and header manipulation, and shows that a further 23× reduction in RPC processing time is possible compared to merely targeting payload manipulation. Cerebros also performs dispatch in hardware, thus eliminating CPU involvement altogether.

iPipe proposes a scheduling algorithm to minimize tail latency for microservices offloaded to SmartNICs [53]. Combining affinity-based request steering with a scheduler such as iPipe's would result in a system that can simultaneously improve instruction locality and maintain tail latency.

RPC Transports. The use of RPCs as a datacenter communication API has resulted in a plethora of research to optimize RPC performance [14, 18, 38, 40, 41, 48, 71]. None of these customized systems use a production RPC layer providing communication among microservices that cross language and data format barriers; therefore, our work is largely orthogonal to all of these systems. Although we chose to implement Cerebros on top of NeBuLa due to its integrated NIC, it is possible to integrate Cerebros with any solution that offers transport termination in hardware. In that case, Cerebros would need to re-implement an interface to the server's memory hierarchy and to the CPU cores, increasing its hardware cost.

Reducing CPU-Accelerator Offload Overhead. Shao et al. have observed that data movement between CPUs and accelerators limits performance, and propose optimizations to pipeline data transfers with computation [65]. Although such techniques could reduce some of the offload overhead we identify, they cannot eliminate it due to the complex pointer dependencies inherent to the messages in production RPC formats. M3-X provides support for accelerators to access OS services and communicate with rescheduled threads [5]. Systems such as Morpheus-SSD [72], GPUfs [66], and NVIDIA GPUDirect [12] address performance losses arising from CPU-mediated transfers between peripherals, and all use peer-to-peer DMA to eliminate CPU time spent moving data through system memory. Cerebros shares the motivation of removing the CPU from the accelerator's path of work, but operates at nanosecond timescales instead of milliseconds.

Many prior works have developed analytical modeling techniques for studying heterogeneous architectures [2, 69]. We instantiate a similar model to show the offload overheads associated with CPU involvement in the flow of RPCs.

Instruction Supply in Servers. A plethora of microarchitectural solutions exist to address instruction supply bottlenecks [3, 22, 44, 45, 50, 51, 63], all of which depend on storing and accessing prefetching metadata. For microservices with many functions or large working sets, the required metadata to cover their misses will outgrow the CPU's storage capacity and reduce coverage. Offloading the RPC layer to Cerebros benefits these frontend designs, as the RPC layer's code footprint vanishes and fewer capacity misses occur in the prefetcher's storage. Cerebros also goes further by proposing affinity-based request steering, which provides speedups in the case where a microservice's functions are too large to be contained by the CPU's frontend resources. Profile-guided prefetching proposals, such as AsmDB [6] and I-SPY [47], perform offline analysis on datacenter-wide miss traces, and re-compile the profiled applications with software prefetches. Affinity-based request steering does not require recompilation or datacenter-wide profiling.

8 CONCLUSION

As the microservices software architecture continues to proliferate, the common RPC layer gluing the microservices together is becoming a bottleneck: the RPC layer itself consumes 40–90% of the execution cycles of the microservices we study. Due to the fact that CPUs are unable to perform the RPC layer's underlying functionality at rates matching commodity NICs, it is necessary to execute the RPC layer in hardware to ensure servers keep pace with improving network line rates. In this work, we present design principles and constraints guiding the architecture of RPC processing hardware. Specifically, we show that an RPC processor must directly receive tasks from the NIC and execute the full RPC layer to completion before the CPU is involved. Following these principles, we propose Cerebros, a NIC-integrated RPC processor that executes the Apache Thrift RPC layer 37–64× faster than a CPU, and also improves the CPU's performance when executing the microservice by improving its instruction supply. Offloading the RPC layer to Cerebros shrinks the microservice's instruction working set by 27–68%, and our novel affinity-based request steering policy provides a further 1.05–2× reduction in execution time for microservices whose functions contend for cache space. We believe Cerebros is an ideal candidate for inclusion in future server chips to support microservices as they decompose into even finer granularity.

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers for their valuable comments and feedback. We also thank the members of PARSA at EPFL for their feedback and support. This work was partially supported by the SNSF "Memory-Centric Server Architecture for Datacenters" and "Hardware/Software Co-Design for In-Memory Services" projects, by Facebook under the "Full-System Accelerated & Secure ML Collaborative Research" program, by the National Science Foundation under the award NSF-CCF-2006602, and by a Google Faculty Research Award.

REFERENCES

[1] Mohammad Alizadeh, Albert G. Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM 2010 Conference.* 63–74.

[2] Muhammad Shoaib Bin Altaf and David A. Wood. 2017. LogCA: A High-Level Performance Model for Hardware Accelerators. In *Proceedings of the 44th International Symposium on Computer Architecture (ISCA).* 375–388.

[3] Ali Ansari, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. 2020. Divide and Conquer Frontend Bottleneck. In *Proceedings of the 47th International Symposium on Computer Architecture (ISCA).* 65–78.

[4] Apache Software Foundation. [n.d.]. *Thrift.* Retrieved August 16, 2019 from https://thrift.apache.org/

[5] Nils Asmussen, Michael Roitzsch, and Hermann Härtig. 2019. M^3x: Autonomous Accelerators via Context-Enabled Fast-Path Communication. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC).* 617–632.

[6] Grant Ayers, Nayana Prasad Nagendra, David I. August, Hyoun Kyu Cho, Svilen Kanev, Christos Kozyrakis, Trivikram Krishnamurthy, Heiner Litz, Tipp Moseley, and Parthasarathy Ranganathan. 2019. AsmDB: understanding and mitigating front-end stalls in warehouse-scale computers. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA).* 462–473.

[7] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. 2013. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition.* Morgan & Claypool Publishers.

[8] Luiz André Barroso, Mike Marty, David A. Patterson, and Parthasarathy Ranganathan. 2017. Attack of the killer microseconds. *Commun. ACM* 60, 4 (2017), 48–54.

[9] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. 2016. A cloud-scale acceleration architecture. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).* 7:1–7:13.

[10] Adrian Cockcroft. 2015. Microservices the Good Bad and the Ugly. Retrieved August 16, 2019 from https://www.slideshare.net/adriancockcroft/microservices-the-good-bad-and-the-ugly

[11] James Coleman. 2009. *Reducing Interrupt Latency Through the Use of Message Signaled Interrupts.* Retrieved March 28, 2020 from https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/msg-signaled-interrupts-paper.pdf

[12] NVIDIA Corp. 2020. *Developing a Linux Kernel Module using GPUDirect RDMA.* Retrieved March 29, 2020 from https://docs.nvidia.com/cuda/gpudirect-rdma/index.html

[13] Alexandros Daglis, Stanko Novakovic, Edouard Bugnion, Babak Falsafi, and Boris Grot. 2015. Manycore network interfaces for in-memory rack-scale computing. In *Proceedings of the 42nd International Symposium on Computer Architecture (ISCA).* 567–579.

[14] Alexandros Daglis, Mark Sutherland, and Babak Falsafi. 2019. RPCValet: NI-Driven Tail-Aware Balancing of µs-Scale RPCs. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXIV).* 35–48.

[15] Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs, Dima Rubinstein, Enrique Cauich Zermeno, Erik Rubow, James Alexander Docauer, Jesse Alpert, Jing Ai, Jon Olson, Kevin DeCabooter, Marc de Kruijf, Nan Hua, Nathan Lewis, Nikhil Kasinadhuni, Riccardo Crepaldi, Srinivas Krishnan, Subbaiah Venkata, Yossi Richter, Uday Naik, and Amin Vahdat. 2018. Andromeda: Performance, Isolation, and Velocity at Scale in Cloud Network Virtualization. In *Proceedings of the 15th Symposium on Networked Systems Design and Implementation (NSDI).* 373–387.

[16] Datacenter Knowledge. 2018. The Year of 100GbE in Data Center Networks. Retrieved November 19, 2020 from https://www.datacenterknowledge.com/networks/year-100gbe-data-center-networks

[17] DPDK [n.d.]. Data Plane Development Kit. https://www.dpdk.org

[18] Aleksandar Dragojevic, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast Remote Memory. In *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI).* 401–414.

[19] Dave Dunning, Greg J. Regnier, Gary L. McAlpine, Don Cameron, Bill Shubert, Frank Berry, Anne Marie Merritt, Ed Gronke, and Chris Dodd. 1998. The Virtual Interface Architecture. *IEEE Micro* 18, 2 (1998), 66–76.

[20] Haggai Eran, Lior Zeno, Maroun Tork, Gabi Malka, and Mark Silberstein. 2019. NICA: An Infrastructure for Inline Acceleration of Network Applications. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC).* 345–362.

[21] Facebook Inc. [n.d.]. *Facebook Thrift.* Retrieved November 19, 2020 from https://github.com/facebook/fbthrift

[22] Michael Ferdman, Thomas F. Wenisch, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. 2008. Temporal instruction fetch streaming. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).* 1–10.

[23] 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 Symposium on Networked Systems Design and Implementation (NSDI).* 51–66.

[24] 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-XXIV).* 3–18.

[25] Google. [n.d.]. *FlatBuffers.* Retrieved April 5, 2019 from https://google.github.io/flatbuffers/

[26] Google. [n.d.]. *gRPC.* Retrieved April 16, 2021 from https://grpc.io/

[27] Albert G. Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. 2009. VL2: a scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM 2009 Conference.* 51–62.

[28] Boris Grot, Joel Hestness, Stephen W. Keckler, and Onur Mutlu. 2009. Express Cube Topologies for on-Chip Interconnects. In *Proceedings of the 15th IEEE Symposium on High-Performance Computer Architecture (HPCA).* 163–174.

[29] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. 2016. RDMA over Commodity Ethernet at Scale. In *Proceedings of the ACM SIGCOMM 2016 Conference.* 202–215.

[30] Tom Halfhill. 2015. Oracle Shrinks Sparc M7. *Linley Group Microprocessor Report* (September 2015).

[31] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. 2017. Re-architecting datacenter networks and stacks for low latency and high performance. In *Proceedings of the ACM SIGCOMM 2017 Conference.* 29–42.

[32] Todd Hoff. 2016. Lessons Learned From Scaling Uber To 2000 Engineers, 1000 Services, And 8000 Git Repositories. Retrieved August 16, 2019 from http://highscalability.com/blog/2016/10/12/lessons-learned-from-scaling-uber-to-2000-engineers-1000-ser.html

[33] Stephen Ibanez, Alex Mallery, Serhat Arslan, Theo Jepsen, Muhammad Shahbaz, Changhoon Kim, and Nick McKeown. 2021. The nanoPU: A Nanosecond Network Stack for Datacenters. In *Proceedings of the 15th Symposium on Operating System Design and Implementation (OSDI).* 239–256.

[34] Intel. 2014. *Introduction to Intel Ethernet Flow Director and Memcached Performance.* https://www.intel.com/content/www/us/en/ethernet-products/converged-network-adapters/ethernet-flow-director.html

[35] Intel Corp. 2016. Intel Xeon Processor D-1500 Product Family. https://cdrdv2.intel.com/v1/dl/getcontent/333423. (Date retrieved: 6 March 2020).

[36] Jaeyoung Jang, Sungjun Jung, Sunmin Jeong, Jun Heo, Hoon Shin, Tae Jun Ham, and Jae W. Lee. 2020. A Specialized Architecture for Object Serialization with Applications to Big Data Analytics. In *Proceedings of the 47th International Symposium on Computer Architecture (ISCA).* 322–334.

[37] Gopal Kakivaya, Lu Xun, Richard Hasha, Shegufta Bakht Ahsan, Todd Pfleiger, Rishi Sinha, Anurag Gupta, Mihail Tarta, Mark Fussell, Vipul Modi, Mansoor Mohsin, Ray Kong, Anmol Ahuja, Oana Platon, Alex Wun, Matthew Snider, Chacko Daniel, Dan Mastrian, Yang Li, Aprameya Rao, Vaishnav Kidambi, Randy Wang, Abhishek Ram, Sumukh Shivaprakash, Rajeet Nair, Alan Warwick, Bharat S. Narasimman, Meng Lin, Jeffrey Chen, Abhay Balkrishna Mhatre, Preetha Subbarayalu, Mert Coskun, and Indranil Gupta. 2018. Service fabric: a distributed platform for building microservices in the cloud. In *Proceedings of the 2018 EuroSys Conference.* 33:1–33:15.

[38] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2014. Using RDMA efficiently for key-value services. In *Proceedings of the ACM SIGCOMM 2014 Conference.* 295–306.

[39] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. Design Guidelines for High Performance RDMA Systems. In *Proceedings of the 2016 USENIX Annual Technical Conference (ATC).* 437–450.

[40] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *Proceedings of the 12th Symposium on Operating System Design and Implementation (OSDI).* 185–201.

[41] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2019. Datacenter RPCs can be General and Fast. In *Proceedings of the 16th Symposium on Networked Systems Design and Implementation (NSDI).* 1–16.

[42] Svilen Kanev, Juan Pablo Darago, Kim M. Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David M. Brooks. 2016. Profiling a Warehouse-Scale Computer. *IEEE Micro* 36, 3 (2016), 54–59.

[43] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, Qijing Huang, Kyle Kovacs, Borivoje Nikolic, Randy H. Katz, Jonathan Bachrach, and Krste Asanovic. 2018. FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud. In *Proceedings of the 45th International Symposium on Computer Architecture (ISCA)*. 29–42.

[44] Cansu Kaynak, Boris Grot, and Babak Falsafi. 2013. SHIFT: shared history instruction fetch for lean-core server processors. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 272–283.

[45] Cansu Kaynak, Boris Grot, and Babak Falsafi. 2015. Confluence: unified instruction supply for scale-out servers. In *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 166–177.

[46] Kenton Varda, Sandstorm.io. [n.d.]. *Cap'n Proto*. Retrieved September 3, 2021 from https://capnproto.org

[47] Tanvir Ahmed Khan, Akshitha Sriraman, Joseph Devietti, Gilles Pokam, Heiner Litz, and Baris Kasikci. 2020. I-SPY: Context-Driven Conditional Instruction Prefetching with Coalescing. In *Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 146–159.

[48] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. 2019. R2P2: Making RPCs first-class datacenter citizens. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*. 863–880.

[49] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan M. G. Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, David Wetherall, and Amin Vahdat. 2020. Swift: Delay is Simple and Effective for Congestion Control in the Datacenter. In *Proceedings of the ACM SIGCOMM 2020 Conference*. 514–528.

[50] Rakesh Kumar, Boris Grot, and Vijay Nagarajan. 2018. Blasting through the Front-End Bottleneck with Shotgun. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXIII)*. 30–42.

[51] Rakesh Kumar, Cheng-Chieh Huang, Boris Grot, and Vijay Nagarajan. 2017. Boomerang: A Metadata-Free Architecture for Control Flow Delivery. In *Proceedings of the 23rd IEEE Symposium on High-Performance Computer Architecture (HPCA)*. 493–504.

[52] 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 *ASPLOS 2021*. 36–51.

[53] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. 2019. Offloading distributed applications onto smartNICs using iPipe. In *Proceedings of the ACM SIGCOMM 2019 Conference*. 318–333.

[54] Pejman Lotfi-Kamran, Boris Grot, Michael Ferdman, Stavros Volos, Yusuf Onur Koçberber, Javier Picorel, Almutaz Adileh, Djordje Jevdjic, Sachin Idgunji, Emre Özer, and Babak Falsafi. 2012. Scale-out processors. In *Proceedings of the 39th International Symposium on Computer Architecture (ISCA)*. 500–511.

[55] Tony Mauro. 2015. Adopting Microservices at Netflix: Lessons for Architectural Design. Retrieved August 16, 2019 from https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices

[56] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John K. Ousterhout. 2018. Homa: a receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the ACM SIGCOMM 2018 Conference*. 221–235.

[57] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. 2014. Scale-out NUMA. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XIX)*. 3–18.

[58] Parallel Systems Architecture Lab (PARSA), EPFL. 2020. QFlex. https://qflex.epfl.ch

[59] Arash Pourhabibi. 2021. Hardware-Software Co-Design of an RPC Processor. *EPFL PhD Thesis* (2021).

[60] Arash Pourhabibi, Siddharth Gupta, Hussein Kassir, Mark Sutherland, Zilu Tian, Mario Paulo Drumond, Babak Falsafi, and Christoph Koch. 2020. Optimus Prime: Accelerating Data Transformation in Servers. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXV)*. 1203–1216.

[61] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John K. Ousterhout. 2018. Arachne: Core-Aware Thread Management. In *Proceedings of the 13th Symposium on Operating System Design and Implementation (OSDI)*. 145–160.

[62] Deepti Raghavan, Philip Alexander Levis, Matei Zaharia, and Irene Zhang. 2021. Breakfast of champions: towards zero-copy serialization with NIC scatter-gather. In *Proceedings of The 18th Workshop on Hot Topics in Operating Systems (HotOS-XVIII)*. 199–205.

[63] Glenn Reinman, Brad Calder, and Todd M. Austin. 1999. Fetch Directed Instruction Prefetching. In *Proceedings of the 32nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 16–27.

[64] Stephen M. Rumble, Diego Ongaro, Ryan Stutsman, Mendel Rosenblum, and John K. Ousterhout. 2011. It's Time for Low Latency. In *Proceedings of The 13th Workshop on Hot Topics in Operating Systems (HotOS-XIII)*.

[65] Yakun Sophia Shao, Sam Likun Xi, Vijayalakshmi Srinivasan, Gu-Yeon Wei, and David M. Brooks. 2016. Co-designing accelerators and SoC interfaces using gem5-Aladdin. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 48:1–48:12.

[66] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. 2013. GPUfs: integrating a file system with GPUs. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XVIII)*. 485–498.

[67] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. 2015. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *Proceedings of the ACM SIGCOMM 2015 Conference*. 183–197.

[68] James E. Smith. 1984. Decoupled Access/Execute Computer Architectures. *ACM Trans. Comput. Syst.* 2, 4 (1984), 289–308.

[69] Akshitha Sriraman and Abhishek Dhanotia. 2020. Accelerometer: Understanding Acceleration Opportunities for Data Center Overheads at Hyperscale. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXV)*. 733–750.

[70] Akshitha Sriraman and Thomas F. Wenisch. 2018. μTune: Auto-Tuned Threading for OLDI Microservices. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*. 177–194.

[71] Mark Sutherland, Siddharth Gupta, Babak Falsafi, Virendra J. Marathe, Dionisios N. Pnevmatikatos, and Alexandros Daglis. 2020. The NEBULA RPC-Optimized Architecture. In *Proceedings of the 47th International Symposium on Computer Architecture (ISCA)*. 199–212.

[72] Hung-Wei Tseng, Qianchen Zhao, Yuxiao Zhou, Mark Gahagan, and Steven Swanson. 2016. Morpheus: Creating Application Objects Efficiently for Heterogeneous Computing. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA)*. 53–65.

[73] Bob Wheeler. 2011. Calxeda Spins 4W Server-on-a-Chip. *Linley Group Microprocessor Report* (November 2011).

[74] Adam Wolnikowski, Stephen Ibanez, Jonathan Stone, Changhoon Kim, Rajit Manohar, and Robert Soulé. 2021. Zerializer: towards zero-copy serialization. In *Proceedings of The 18th Workshop on Hot Topics in Operating Systems (HotOS-XVIII)*. 206–212.

[75] Hao Zhou, Ming Chen, Qian Lin, Yong Wang, Xiaobin She, Sifan Liu, Rui Gu, Beng Chin Ooi, and Junfeng Yang. 2018. Overload Control for Scaling WeChat Microservices. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2018,Carlsbad, CA, USA, October 11-13, 2018*. 149–161. https://doi.org/10.1145/3267809.3267823