AccelFlow: Orchestrating an On-Package Ensemble of Fine-Grained Accelerators for Microservices

Jovan Stojkovic
University of Illinois at Urbana-Champaign
Urbana, Illinois, USA
jovans2@illinois.edu

Abraham Farrell
University of Illinois at Urbana-Champaign
Urbana, Illinois, USA
af28@illinois.edu

Zhangxiaowen Gong
Intel Corporation
Santa Clara, California, USA
zhangxiaowen.gong@intel.com

Christopher J. Hughes
Intel Corporation
Santa Clara, California, USA
christopher.j.hughes@intel.com

Josep Torrellas
University of Illinois at Urbana-Champaign
Urbana, Illinois, USA
torrella@illinois.edu

***Abstract*—Microservices suffer from the execution of auxiliary operations known as datacenter tax, such as RPC and TCP processing, and data (de)serialization, (de)encryption, and (de)compression. To minimize this tax, multiple hardware accelerators have been proposed. However, it is unclear how these accelerators should be orchestrated. Past work has focused only on orchestrating accelerators in coarse-grained environments with monolithic applications.**

In this paper, we characterize the needs of orchestrating an ensemble of on-package accelerators in microservice environments. We observe that orchestration frameworks need to be highly dynamic and nimble. The basic operations to be accelerated are fine grained, potentially taking only tens of μs. Moreover, the sequence of accelerators to use is often affected by "branch conditions" whose real-time resolution determines the set of subsequent accelerators needed. To address these challenges, we present *AccelFlow*, the first orchestration framework for on-package accelerators of microservices. In AccelFlow, CPU cores build software structures called *Traces* that contain sequences of accelerators to call. A core enqueues a trace in an accelerator in user mode and, from then on, the accelerators in the trace execute in sequence without CPU involvement. A trace can include branch conditions whose outcomes determine the trace control flow. Compared to state-of-the-art accelerator orchestrators, AccelFlow on average reduces P99 tail latency by 70%, reduces average latency by 38%, and increases throughput by 120%.

I. Introduction

Datacenter workloads increasingly use the microservice paradigm [23], [65], where an application is divided into multiple services that are deployed as separate programs communicating with each other. With microservices, applications scale more easily, are easy to deploy and maintain, and can use diverse programming languages and frameworks. As a result, cloud service providers have embraced this technology [10], [23], [29], [54], [55], [61], [78]–[80], [88].

Unfortunately, microservice environments suffer from various software overheads [19], [42], [71], [72]. For instance, microservices are commonly implemented as Remote Procedure Call (RPC) servers [3], [24], which facilitates their distribution across machines and independent scalability. However, RPC processing has overhead. Further, data communicated between services must undergo (de)serialization to ensure compatibility across different programming languages through standardized protocols [21]. Additionally, microservice environments use encryption for security and compression to reduce resource use. Collectively, these auxiliary or "glue" operations are known as *datacenter tax* [42], and can consume a substantial fraction of CPU cycles in datacenters [20], [42], [70].

To minimize datacenter tax, researchers have proposed numerous hardware accelerators or software techniques that target a single source of datacenter tax [1], [7], [28], [30], [33], [34], [38], [39], [43]–[45], [50], [63], [84]—e.g., compression, Transmission Control Protocol (TCP), or RPC processing. Given the many sources of datacenter tax [20], these proposals must be augmented with a way to efficiently orchestrate the multiple accelerators of an ensemble of accelerators integrated in a server. Efficiency is key because, in microservice environments, the operations to be accelerated may take only tens of microseconds.

There are proposals to orchestrate on-package accelerators for image/video processing or ML *in monolithic applications* [12], [26], [27], [60], [82]. In some of these proposals, a CPU core [27] or a centralized hardware manager [12], [26] orchestrates the accelerators by invoking them and receiving an interrupt when each accelerator completes. This approach introduces major coordination overheads. Two designs allow direct accelerator-to-accelerator communication. One of them [82] statically links some pairs of accelerators but otherwise relies on the cores to orchestrate the accelerators. The other design [60] fixes the sequence of accelerators, mimicking a pipelined engine. All these schemes are suitable for coarse-grained accelerator operations or static environments. They are unsuitable for environments where we want to accelerate fine-grained, highly-varying sequences of operations.

Our analysis shows that past approaches are insufficient for microservices. Microservices need sequences of accelerators that are *fine-grained* and *vary widely* both across services and across invocations of the same service. Moreover, the sequence of accelerators to invoke is often affected by conditions whose real-time resolution determines the set of subsequent accelerators needed. Additionally, the data transferred between accelerators may vary in format. All this demands a *dynamic and flexible* orchestration framework.

To address these challenges, we propose *AccelFlow*, the first accelerator orchestration framework for microservices. AccelFlow orchestrates an on-package ensemble of fine-grained accelerators without coordination from a CPU or a centralized hardware manager. AccelFlow introduces the concept of *Traces of Accelerators*: software structures built by cores that contain a sequence of accelerator IDs. A CPU core triggers accelerator execution by enqueuing a trace in an accelerator in user mode. From then on, the accelerators in the trace execute in sequence without CPU involvement, passing data from one accelerator to the next. Moreover, AccelFlow introduces the idea of *Branch Conditions* in a trace, which are conditions resolved on-the-fly without CPU involvement, and whose outcomes determine the control flow inside the trace.

Each accelerator in AccelFlow has a standard interface with input and output hardware queues, and input and output controllers (called dispatchers). The input dispatcher schedules requests enqueued in the accelerator. The output dispatcher computes any branch condition in the trace and sends the accelerator's output data to the input queue of the next accelerator in the trace using a DMA engine.

We evaluate AccelFlow with full-system simulations of a server with an Intel IceLake-like processor and nine state-of-the-art accelerators. We run large open-source microservice applications [19] with real-world invocation traces [54]. Our results show that AccelFlow is very effective. Compared to state-of-the-art proposals for accelerator orchestration [26], [82], on average across services, AccelFlow reduces P99 tail latency by 70%, reduces average latency by 38%, and increases throughput by 120%.

This paper makes the following contributions:

- A characterization of how microservice environments can use ensembles of on-package accelerators.
- AccelFlow, the first orchestration framework for an ensemble of fine-grained accelerators in microservice environments.
- Evaluation of AccelFlow, compared to the state-of-the-art.

II. Background

Typical Workflow of a Microservice Invocation. A microservice invocation arrives to the server as an encrypted network message. To achieve reliable and ordered delivery of messages, the TCP stack [40] first performs message reassembly, congestion control, and checksum calculation. Then, the SSL protocol [62] is invoked to authenticate the client and decrypt the message using algorithms such as RSA, AES, or SHA.

Next, the RPC stack [3], [24] processes the decrypted message, which contains the name of the function to invoke and its arguments. A microservice may have multiple entry points or functions that users can invoke. The server keeps a table that maps the name of the function to its *handler* and *descriptor*. The role of the RPC stack is to decode the name of the function from the message and fetch the function handler and descriptor from the table. Then, a deserialization protocol such as Protobuf [21] takes the function handler, descriptor, and serialized arguments. Protobuf uses the function descriptor to deserialize the arguments—i.e., to translate the arguments' wire format to their application format in a given language. To reduce the network bandwidth use, large arguments may come compressed, e.g., with Zstd [17] or Snappy [22]. Thus, after deserialization, these arguments are decompressed.

Finally, the invocation is ready to execute. Using a software or hardware algorithm, a load balancer [33] picks a core to execute the function. When execution completes, the steps above are performed in reverse order: compress the results, serialize the results, encode the message in the RPC stack, encrypt the message, and transmit it via TCP. Within an invocation, the function may also invoke other services or access storage, creating nested RPCs with similar steps.

CPUs with Integrated Accelerators. CPU vendors have long integrated special-purpose hardware to accelerate common functions—e.g., the cryptography accelerator in Sun's UltraSPARC T1 [75] or GPUs in AMD's Llano APUs [8]. Recently, Intel integrated several accelerators into their Sapphire Rapids CPU [86], targeting datacenter functions. This CPU introduces enhancements to the system architecture for the accelerators. The accelerators are not PCIe devices programmed via memory-mapped I/O operations. Instead, their ISA has instructions for dispatching work and for signaling; the accelerators operate with virtual addresses exploiting the IOMMU for address translation; and the accelerators support virtualization to make them usable in a cloud environment.

III. Opportunities and Challenges of Accelerator Ensembles for Microservices

To understand the opportunities and challenges of using an ensemble of accelerators in microservice environments, we measure the performance of a 36-core Intel Xeon Platinum server [35] at 2.4GHz, and estimate the potential impact that nine hardware accelerators proposed by prior work could have. These accelerators speed-up TCP [7], (De)Encryption (Decr and Encr) [39], RPC [63], (De)Serialization (Dser and Ser) [44], (De)Compression (Dcmp and Cmp) [45], and load balancing (LdB) [33]. We run over 80 open-source services from DeathStarBench [19], Train Ticket [90], and μSuite [72]. The infrastructure is presented in Section VI. Here, we present our main observations.

***Q1:* Is a Processor with Many Accelerators a Good Environment for Microservices?** Figure 1 breaks down the execution time of SocialNetwork services from DeathStarBench on our server. We identify code sections that could be assigned to one of the accelerators considered, and measure their execution time. The remaining time of the service is called *AppLogic*. A given accelerator category may correspond to multiple code sections. Bars are normalized and the numbers on top of them are the absolute service execution times.

We see that service invocations spend most of their execution time on tax. On average, an invocation spends only 20.7% of its end-to-end execution time on the core application logic. It spends 25.6%, 14.6%, 3.2%, 22.4%, 9.5% and 3.9% of its time on the TCP, (De)Encr, RPC, (De)Ser, (De)Cmp and LdB operations, respectively. The relative weight of tax increases for microservices with (1) short execution times

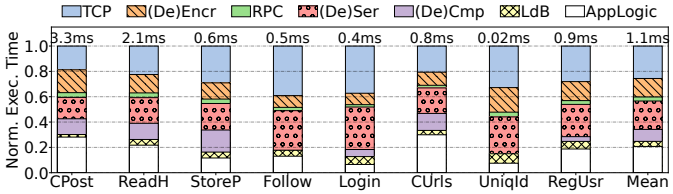

Fig. 1: Execution time breakdown of SocialNetwork service invocations on an Intel Xeon server. The numbers on top of the bars are the absolute execution times of the invocations.

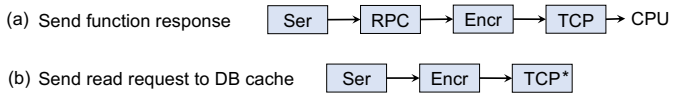

Fig. 2: Examples of sequences of datacenter tax operations.

(e.g., UniqId) or (2) many nested RPCs or remote storage accesses (e.g., Login). Services from other suites are similar. Hyperscalers have also seen that tax dominates the execution of their services [20], [42], [70].

In many cases, multiple tax operations are executed back to back, in a sequence, without interleaved operations of the core logic. Figure 2 shows sequences of tax operations executed when a core sends a function response (Figure 2a) or a read request to a database cache (Figure 2b). The first sequence involves Ser, RPC, Encr, and TCP; the second one has Ser, Encr, and TCP. More details are given later.

If we have accelerators for each of these operations, the question arises as to how to orchestrate them. We consider three approaches. First, in *CPU-Centric* [27], a CPU core invokes one accelerator at a time. When an accelerator completes, it interrupts the core, which can then invoke the next accelerator. Second, in *HW-Manager* [12], [18], [26], the CPU offloads the scheduling of the accelerator requests to a centralized hardware manager. The CPU submits to the manager the sequence of accelerators to invoke. When an accelerator completes its job, it interrupts the manager, which then calls the next accelerator in the chain. On completion of a chain, the manager interrupts the CPU. Finally, in *Direct*, scheduling is performed neither by a CPU core nor by a manager. Instead, a CPU core passes a list of accelerators that need to execute in sequence to the first accelerator. After each accelerator in the sequence executes, it directly calls the next accelerator in the sequence.

We simulate the execution of these three environments, modeling a server like the one measured, and the nine accelerators [7], [33], [39], [44], [45], [63]. For *HW-Manager*, we model RELIEF [26]. We describe the simulation infrastructure in Section VI. We record the time taken by the orchestration overhead—i.e., sending and receiving interrupts, and communicating between CPU core, hardware manager, and accelerators. Figure 3 shows the average orchestration overhead as a fraction of the total execution time of the service. The figure shows data for the three approaches above, as the load of the 36-core processor in kilo-requests per second

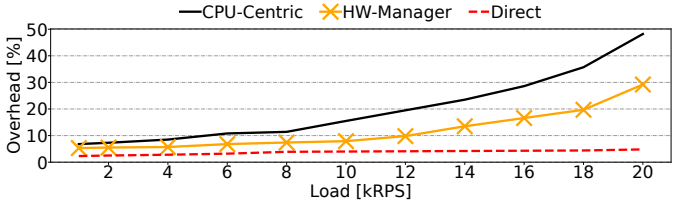

Fig. 3: Orchestration overhead of different approaches averaged across all services with a varying load of services.

TABLE I: Source/destination accelerators for each accelerator.

Accelerator	Src Accelerators	Dst Accelerators
TCP	Ser, Encr, Cmp	LdB, Decr, Dser, Dcmp
Encr	TCP, RPC, Ser	TCP, RPC
Decr	TCP	RPC, Dser
RPC	Decr, Ser	Encr, Deser, LdB
Ser	Deser, Cmp, CPU	TCP, Encr, RPC
Dser	TCP, Decr, RPC	Ser, Dcmp, LdB
Cmp	Deser, CPU	Ser, LdB, CPU, TCP
Dcmp	Deser, TCP, CPU	(De)Ser, LdB, CPU, TCP
LdB	TCP, Dser, Dcmp	CPU

(kRPS) changes. We see that *Direct* has less overhead than *HW-Manager*, which has less overhead than *CPU-Centric*. The overhead of the last two approaches increases rapidly with the load. For 15 kRPS, the overhead in *CPU-Centric* and *HW-Manager* is 25% and 15%, respectively.

***Q2:* What is the Control Flow in Accelerator Sequences?** To understand whether the sequences of accelerators that need no intervening CPU involvement are deterministic, we analyze the sequences in our 80 services. For each accelerator in one such sequence, we record which accelerator provides its input (the source) and which accelerator consumes its output (the destination). As shown in Table I, an accelerator can consume data from and produce data for multiple accelerators. Thus, the inter-accelerator connections need to be flexible.

We now consider the control flow inside accelerator sequences that contain no intervening CPU involvement. We find that a sequence often includes dynamic control flow. Figure 4 shows two examples. Figure 4a is the sequence when a processor receives a function request. The payload may be compressed and require decompression. However, this is unknown until the deserialization step. At that point, depending on the value of a field in the message, we may need to invoke the Dcmp accelerator.

Figure 4b shows the sequence when a processor receives the response for a read request to the database cache. Based on whether the request hit in the cache, different actions are needed. If it hit, the response has the data and needs to be sent to a CPU core; otherwise, the response has no data, and a new request needs to be issued to the actual database. Whether a hit occurred is only known after Dser has completed.

Our data shows that 69.2%, 62.5%, 82.5%, and 53.8% of the sequences of accelerators in the *SocialNet*, *HotelReservation*, and *MediaServices* microservices from DeathStarBench, and *TrainTicket*, respectively, have at least one conditional statement. Some sequences have up to four. Given this, interrupting

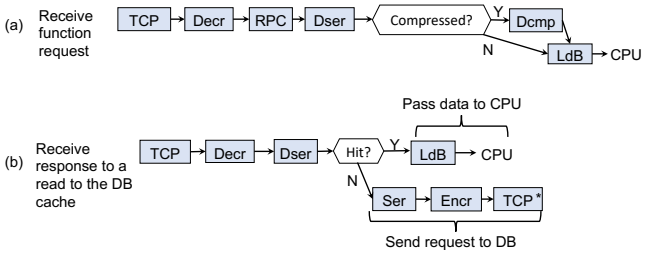

Fig. 4: Dynamic control flow in sequences of tax operation.

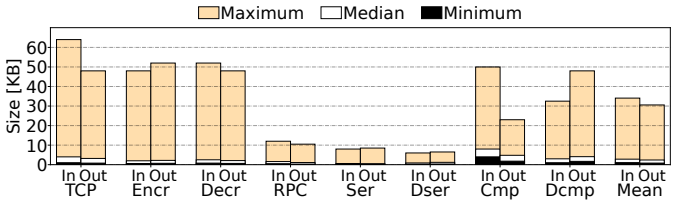

Fig. 5: Sizes of the input/output data of each accelerator.

the CPU every time that a sequence of accelerators encounters a conditional statement would induce substantial overhead. These conditionals are simple, as they involve checking a few bits in the payload, and performing simple comparisons and computation logic.

***Q3:* How Should Data be Forwarded Between Accelerators?** The data forwarded from one accelerator to the next one can vary in size and in format. Figure 5 shows the maximum, median, and minimum size of the input and output data of each accelerator. There is no LdB bar because LdB does not process data; it simply picks a core to process the request. As shown in the figure, the median data size is small, i.e., a few KBs (as also observed by Google [68]). There is, however, a long tail with a few tens of KB. Further, in some cases, the data transferred between two accelerators is generated in one format is consumed in a different format. Typically, the transformations required are simple, such as transforming from string to BSON format [59].

Summary. Microservices can benefit from ensembles of accelerators that target different sources of datacenter tax. These accelerators can be invoked in sequences without an intervening CPU. For highest performance, the orchestration of such ensembles should avoid invoking a CPU or a centralized hardware manager. Instead, within a sequence, accelerators should handle dynamic control flow, data transformations, and data transfers of different sizes.

IV. AccelFlow: Orchestrating Accelerators

Based on this analysis, we propose the *AccelFlow* architecture to orchestrate ensembles of accelerators for microservices. This section describes its hardware organization, execution model, and other aspects.

A. AccelFlow Hardware Organization

We envision *AccelFlow* to be implemented in a large, multi-chiplet processor with many cores and one or more instances of all the accelerators of Section III. The processor can be

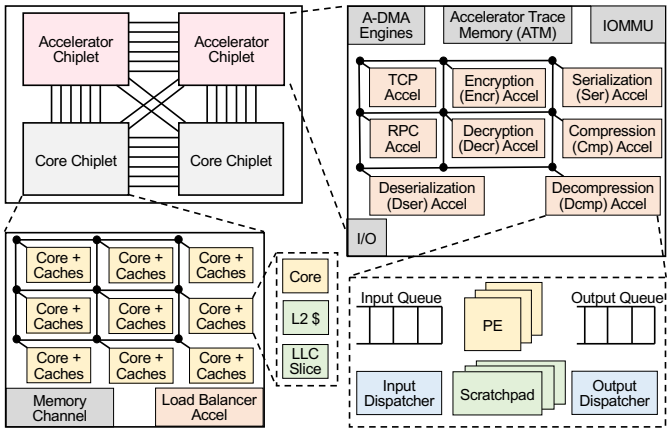

Fig. 6: Possible organization of a processor with AccelFlow.

organized in different ways: cores and accelerators may share the same chiplet like in the Intel Sapphire Rapids (SPR) [37], or accelerators may be in their own chiplet—e.g., the I/O die of some AMD and Intel servers. The latter design is the one we assume and is shown in Figure 6: one type of chiplet contains an instance of all the accelerators except for LdB, and another type of chiplet contains cores, private caches, a distributed shared LLC and LdB. The LdB accelerator is in the core chiplet since it is tightly coupled with the cores.

Our design assumes that all the accelerators have the same standard interface. This is a futuristic goal that is motivated by simplicity and by recent industry trends, such as the Intel Accelerator Interfacing Architecture (AiA) [49]. Moreover, accelerators and cores share the same virtual address space (like Intel's Shared Virtual Memory (SVM) [48] in SPR). Like in SPR, the accelerators directly read from and write to the processor's LLC in a cache coherent manner—i.e., on a read, they get the correct data in the system, while on a write, they invalidate all private caches.

Sequence of Accelerators or *Trace*. In AccelFlow, CPU cores build software structures called *Traces* that contain a sequence of accelerator IDs, and store them in a special on-chip memory called *Accelerator Trace Memory* (ATM) (Figure 6). A trace can include branch conditions, whose outcomes determine the control flow inside the trace. Branches are encoded as simple logic operations on bits within the payload of the message—e.g., "if (field1 & field2) goto Ser; else goto Cmp". A trace may also contain data transformation fields and, in its tail, the address in the ATM that contains the next trace to execute after this one completes.

A CPU core triggers accelerator execution by passing a trace to an accelerator. From then on, the accelerators in the trace will execute in sequence without CPU involvement, passing data from one accelerator to the next, potentially executing branch conditions, and potentially accessing the ATM for additional traces. Once the trace(s) have executed, control returns to the initiating CPU core.

Architecture of an Accelerator. An accelerator has an SRAM input queue and an SRAM output queue, an input and an output controller (called *Dispatchers*), and multiple processing

elements (PEs), each one with a scratchpad, that perform the same operations (Figure 6). A scratchpad contains accelerator PE state that may be private to a request (e.g., the History Window [45] for Cmp/Dcmp), or shared by all requests of the same service (e.g., the Accelerator Descriptor Table (ADT) [44] for Ser).

Each queue entry contains multiple fields. The main ones are: (1) space for a trace with a moving *Position Mark* that indicates which accelerator is to execute next, (2) the ID of the tenant that is using the entry (since an accelerator can be used by multiple tenants), (3) up to 2KB of data to be operated upon, to capture the common-case data sizes, and (4) the virtual address of a *Memory Pointer* that points to a software buffer in memory where more data is stored. Large inputs or outputs (>2KB) store some data in these locations. Such a buffer is cacheable in the cores' caches, and is kept cache coherent. The input and output hardware queues are not visible to the cache coherence protocol.

The input dispatcher manages the input queue, deciding which entry should be executed next by which PE. If an input queue entry needs data from multiple source accelerators, the dispatcher marks the entry as ready when all data has arrived.

After a PE deposits its output in the output queue, the output dispatcher performs multiple operations. First, if the next field in the trace is a branch condition, it resolves the condition. Second, if the next trace field includes a data transformation, it transforms the output data. Third, it uses one of a set of on-chip DMA engines (*A-DMA* in Figure 6) to send the output data to the input queue of the next accelerator in the sequence. Finally, after the last accelerator in the trace has executed, the output dispatcher examines the tail of the trace. If it finds the address of an ATM location, it accesses it to get the next trace to run. Otherwise, it notifies a CPU core.

Operation of an Accelerator. PEs in an accelerator consume input queue entries as mandated by the input dispatcher. A PE consumes the data from an input queue entry plus any additional data in the CPU memory hierarchy that is accessible through the Memory Pointer in the input queue entry. A PE is non-preemptible: it runs tasks to completion. After the PE completes the operation, it deposits the data results and all the metadata (including the trace) in an entry of the output queue. The output dispatcher handles the entry from then on.

Accelerators exploit PCIe's Address Translation Service (ATS) [4]. A device submits address translation requests to the IOMMU (Figure 6). An ATS request converts a process ID and virtual address from that process to a physical address. Each accelerator caches the results in an address translation cache (a TLB) that is also accessed by the input/output dispatchers.

On an exception, the accelerator operation stops, a CPU core is interrupted, and the OS handles the exception.

Anatomy of the Execution of Traces. When the software wants the accelerator ensemble to execute a sequence of operations, it creates a trace. Then, a CPU core executes an *Enqueue* instruction in *user mode*. Enqueue takes as arguments an accelerator ID (the first accelerator in the trace) and the trace. The instruction triggers the input dispatcher of the corresponding accelerator to allocate an entry in the input queue and store the trace in the entry. Enqueue returns the index of the input queue entry. Then, the core invokes an A-DMA engine, passing the accelerator ID, the input queue index, and a pointer to the payload data. The A-DMA engine uses the pointer to coherently collect the data and deposit it in the corresponding input queue entry. If the Enqueue instruction returns an error (e.g., no space in the queue), the core retries with another accelerator of the same type.

Execution proceeds from accelerator to accelerator, transforming the data, and moving the data and trace from the output queue of one to the input queue of another. When the last accelerator of the trace completes its execution, its output dispatcher uses an A-DMA engine to move the resulting data to a memory location. The dispatcher then sends a *user-level* notification to the CPU core that initiated the ensemble execution, passing the virtual address of the memory location that contains the result. To reduce overhead, such notification is not an interrupt. The CPU core can be performing other work and periodically check the notification flag. Alternatively, the CPU core can wait for the notification by continuously polling the flag, or by executing an MWAIT-like instruction [31] and automatically waking up on notification.

Sometimes, it is desirable to partition a trace into multiple subtraces. This is done for three reasons: to avoid transferring a very long trace, to enable the reuse of individual subtraces across multiple operations, or when, at a point in the trace, a decision needs to be made that can cause a major divergence in the set of accelerators to invoke next. In these cases, the software creates multiple traces to execute in sequence. Before the CPU core invokes the first trace, it stores all the subsequent traces in the ATM. Moreover, in the tail of each trace but the last one, it places the address of the next trace in the ATM.

With this design, when an output dispatcher processing a trace finds that the trace concludes with an ATM address, the output dispatcher reads the next trace from the ATM and deposits it in the input queue of the next accelerator.

Since there are nine accelerator types, we use 4 bits per accelerator in the trace. We set the maximum size of a trace to be 8 bytes, which allows up to 16 accelerator invocations per trace. If a sequence exceeds 8 bytes, AccelFlow would split it into multiple subtraces. In our evaluation, we do not observe long traces requiring splitting. A major divergence occurs when the trace would include rare events, such as exceptions or errors. We split it to avoid consuming unnecessary bandwidth when moving traces from one accelerator to another. We discuss this case later.

Preventing Starvation & Deadlock. When a core invokes Enqueue, it may need more than one try to find an accelerator with a free input queue entry. If, after multiple attempts, the core cannot find it, trace execution falls back to the core. This eliminates starvation.

Similarly, when an output dispatcher wants to enqueue an entry in an input queue, it may find that the latter is full. Unlike the case for Enqueue, where the CPU gets an error and retries,

the output dispatcher cannot keep retrying. Hence, each input queue is associated with an *Overflow* pointer that points to an overflow area in memory. When an output dispatcher finds a full input queue, it stores the data in the queue's overflow area. The overflowed entries are progressively moved into actual input queue entries as the latter become free. Supporting this design is simplified by the fact that the output dispatcher uses virtual addresses. If the overflow area is full, execution falls back to the core. This avoids deadlock.

B. AccelFlow Execution Model

We now examine AccelFlow's execution model, including trace termination and initiation, and the list of traces.

Terminating a Trace Execution. When a trace execution terminates, AccelFlow can either store the results in memory and notify the CPU core that initiated it (Section IV-A), or it can instead start the execution of another trace stored in the ATM. The latter happens, for example, when the last operation of the trace involves the TCP accelerator sending a message that will induce a response. An example is shown in Figure 2b, which is the trace executed to send a read request to a database cache. The trace invokes the Ser, Encr, and TCP accelerators. The asterisk in the TCP box means that the last entry in the trace includes an ATM address. After TCP sends the request, the TCP output dispatcher accesses the ATM address, which contains the trace to execute on a message reception. This trace is immediately loaded in the input queue of the same TCP accelerator. Later, when the response is received, it is routed to the same TCP (since the initial request included the TCP ID) and triggers the execution of the stored trace.

These input queue entries in a TCP accelerator are not held indefinitely waiting for a response. After a specified timeout, the core is notified and terminates the request.

Triggering the Execution of a Trace. A trace execution may be triggered by a CPU core with an Enqueue (Section IV-A) or by the arrival of a message. An example of the latter is shown in Figure 4b, which is executed when the response of the read to the database cache is received. As indicated above, the TCP that sent the request already has the new trace ready.

On message arrival, the trace invokes the TCP, Decr, and Dser accelerators. The data generated by Dser contains a field that indicates whether the request hit in the database cache and returned the data. The output dispatcher of Dser checks the field. If it is set, the data is passed to the LdB accelerator and then the CPU core that started the request is notified. The trace always contains the ID of the CPU core that started it. That CPU core contains a record of what thread issued the original request and needs to be notified.

If, instead, the field is clear, the request had missed in the cache and a read message must be sent to the DB. Hence, the Ser, Encr, and TCP accelerators are invoked. Then, as denoted by the asterisk in the figure, the ATM address at the tail of the sequence is used to access the ATM. The trace in that location is loaded in the input queue of the same TCP accelerator.

Handling Request Initiation and Completion. Two noteworthy cases are: a core receiving a *new* request for a function and a core sending the final response to the client. To handle the former case, all TCP accelerators already store a trace like the one in Figure 4a. The trace follows the execution of TCP with that of Decr, RPC, Dser, Dcmp (if the data is compressed), and LdB. LdB saves the results and notifies a CPU core. The data saved includes the ID of the TCP.

When the core finally sends the response of the function, the ensemble executes the trace in Figure 2a—after reading it from the ATM or as initiated by a CPU core. The trace triggers the execution of Ser, RPC, Encr, and the same TCP that received the message—since it contains the TCB (transmission control block) of the request in its internal state. After the message is sent, the CPU is notified.

TABLE II: Traces that our services use. DB means database.

Trace	Explanation
T1	Receive function request (with or without Dcmp).
T2	Send function response without Cmp.
T3	Send function response with Cmp.
T4	Send read request to DB cache.
T5	Receive response to a read to the DB cache (with or without Dcmp).
T6	Receive response to a read to the DB (with or without Dcmp or Cmp).
T7	Receive response to a write to the DB cache or DB.
T8	Send write request to DB cache or to DB (with or without Cmp).
T9	Send RPC request (with or without Cmp).
T10	Receive RPC response.
T11	Send HTTP request (with or without Cmp).
T12	Receive HTTP response.

Complete List of Traces in our Services. Table II shows the complete list of traces that we have identified for our services.

T1 and T2 are shown in Figure 4a and 2a, respectively. T3 is like Figure 2a except that a Cmp is invoked before Ser; there is no branch because the CPU core knows that it needs to compress the data. T4 is shown in Figure 2b. T5 is shown in Figure 7, as Figure 4b was a simplified version of it without a check for compressed data and a Dcmp.

T6 is shown in Figure 7. If the data was not found in the DB, the function returns an error; in this case, the error is reported to the user. Otherwise, the data is first potentially decompressed and then, both passed to the CPU and written to the DB cache in parallel. To write to the cache, the data may need to be compressed again with another algorithm if the cache uses compressed data (*C-Compressed* test).

T7 in Figure 7 is executed when the response from the write to the database cache returns. The same trace is also executed when the response to a write to the database is received. The response may include an exception; in this case, the function error is directly reported to the user by the accelerator ensemble. T8-T12 are also shown in Figure 7 and are largely self-explanatory. The response from an RPC (T10) may include an exception that is handled as in T7. In HTTP responses (T12), errors are taken care by the CPU.

As indicated in Section IV-A, in the traces with exceptions or errors (T6, T7, and T10), the infrequently-exercised four-accelerator subsequences that handle these cases are removed and placed in a trace of their own.

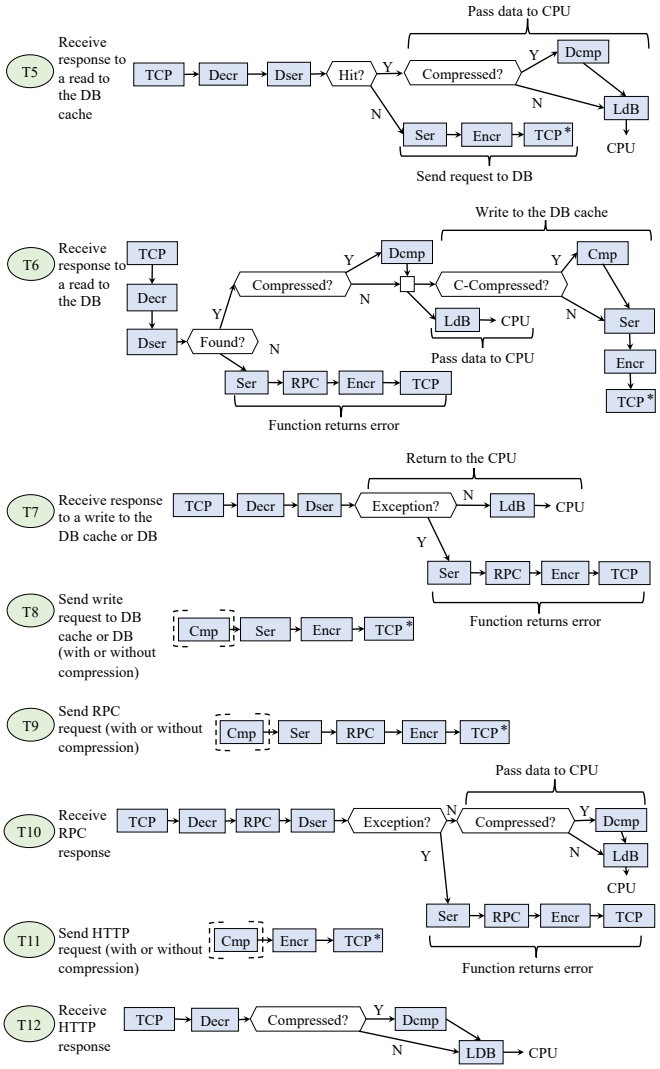

Fig. 7: Additional traces beyond those in Figures 2 and 4.

C. Soft Service Level Objectives (SLOs)

The requests in the input queue of an accelerator are processed in FIFO order or in priority order (if the software has tagged them with priority levels). If the system supports requests under SLOs, AccelFlow is augmented as follows. When a core needs to process a function that has an SLO, as the core generates the DAG of accelerators to invoke, it assigns a deadline to each of the acceleration steps. The deadlines are set by software, and are treated as *soft deadlines*. They may be a function of the function's inputs. These deadlines are stored in the *first* trace that the core provides with Enqueue and are passed, as part of the trace, between accelerators. During execution, these deadlines are relative to the start of the execution: if an accelerator finishes early, it passes the slack on.

When a request gets queued-up in the input queue of an accelerator, the input dispatcher reads its deadline and estimates whether it will meet its deadline. If it will not, the dispatcher checks earlier requests. If any has a large slack, the dispatcher can change the processing order so that both requests meet deadlines. If this is not possible, the dispatcher may allow the request to proceed and record if it ends up violating its SLO.

D. Accelerator Virtualization

Past work has proposed coarse-grain accelerator virtualization. Specifically, in AuRORA [47], a CPU requests the reservation of an accelerator and, when it is granted, uses the accelerator by itself until it completes. In contrast, AccelFlow uses a fine-grain approach to virtualization. When a CPU core issues an Enqueue, it includes the tenant ID (assigned by the VMM) in the trace. The tenant ID is passed with the trace across accelerators. Hence, the input/output queues of an accelerator can have entries from different tenants, and each entry is tagged with the ID of the tenant that owns the data. As the PEs of an accelerator process input entries, the hardware clears the state of a PE and its scratchpad in between executions of entries from different tenants. This enables multiple tenants to securely use the accelerator ensemble concurrently.

To prevent tenants from hoarding accelerators, AccelFlow sets a limit to the number of traces from any individual tenant that can be executing concurrently. For each tenant, when a trace starts, a counter is incremented, and when a trace ends, the counter is decremented. If a counter for a tenant reaches a threshold N, no new trace for the tenant can be initiated. Since, in most cases, a trace only uses one accelerator at a time, this approach ensures that a tenant cannot use more than N accelerators at a time. This design can be combined with a technique that limits memory bandwidth use by a tenant in the memory controller, such as Intel's Memory Bandwidth Allocation (MBA) [36].

V. Detailed AccelFlow Implementation

This section describes the implementation of the input and output dispatchers, the interconnection network, and how to program AccelFlow.

1. Input Dispatcher. The input dispatcher of an accelerator is a Finite State Machine (FSM). Figure 9 shows an input queue, an input dispatcher, and three accelerator PEs. The input dispatcher continuously monitors the *Free?* flags of the PEs in the accelerator and the *Ready?* flags of the entries in the input queue of the accelerator. If a *Ready?* and a *Free?* flag are set, the dispatcher may move the ready entry from the queue into the free PE and clear the entry. If the entry's data is larger than 2KB, the dispatcher obtains the rest of the data by following the *Memory Pointer* field in the entry (Section IV-A). Table III shows the latency and bandwidth of transferring data from the input queue to the scratchpad of a PE. The transfer is pipelined to improve throughput. In addition, since transfers to different PEs use different ports, multiple queue entries can be transferred concurrently.

While the base AccelFlow design processes the input entries in FIFO order, more advanced policies could process the entries based on their *Priority* field (if there are priorities) or *Deadline* field (if the system uses SLOs) (Section IV-C). In this

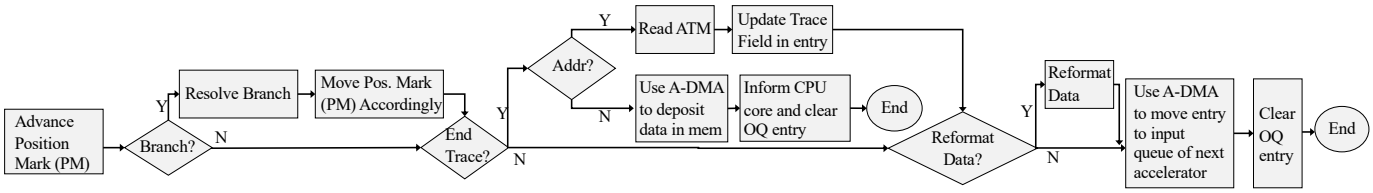

Fig. 8: Flowchart of the output dispatcher operation. *OQ* stands for output queue.

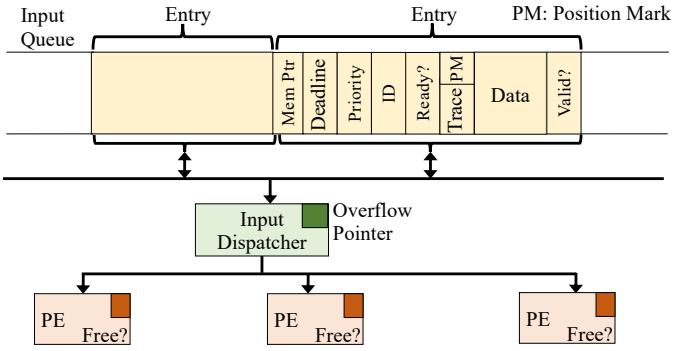

Fig. 9: Input dispatcher and input queue of an accelerator.

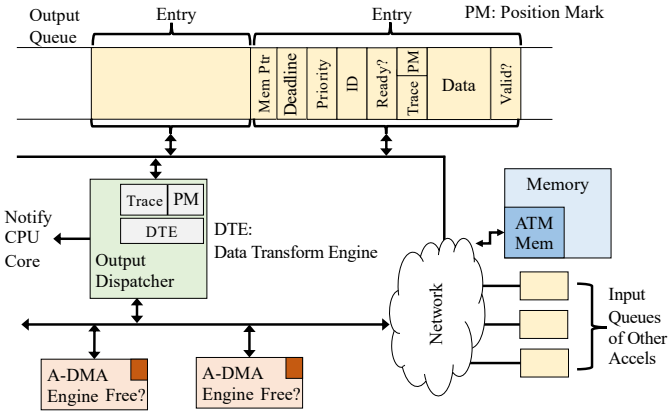

Fig. 10: Output dispatcher and output queue in AccelFlow.

case, the *Priority* and *Deadline* fields are set by the software and are carried by the incoming message. If the input queue *Overflow Pointer* is set, as soon as a queue entry is moved into a PE, the dispatcher follows the Overflow pointer and moves an entry from there into the input queue.

2. Output Dispatcher. The output dispatcher of an accelerator is an FSM. Figure 10 shows an output queue, an output dispatcher, two A-DMA engines, the ATM memory, and the input queues of other accelerators. The output dispatcher continuously monitors the *Ready?* flags of the entries in the output queue of the accelerator.

When an entry with the *Ready?* flag set is found, the dispatcher reads the trace and its Position Mark (PM) into registers (Figure 10). Then, it executes the flowchart of Figure 8. As shown in the flowchart, the dispatcher first advances the PM and checks if the next field is a *Branch* field (which encodes what operations to perform on what fields of the payload). If so, the dispatcher performs these operations on the data in the output queue entry. The computations involve loads/stores and simple ALU operations. Based on the result, the dispatcher advances the PM to the correct place of the trace (Figure 8). It then saves the PM to the output queue entry.

Next, the dispatcher checks if it has reached the end of the trace. If so, it checks if the tail of the trace has an address. If it does, the dispatcher reads the ATM at this address and loads its contents (i.e., the new trace to execute and PM) into both the output queue entry and the dispatcher registers.

If, instead, the end of trace was reached but there was no address, the dispatcher finds an A-DMA engine with a *Free?* flag set and programs it to move the payload data from the *Data* field in the output queue entry to a memory location (Figure 8). Once the engine completes, the dispatcher informs the CPU core and clears the output queue entry.

In the cases when we are not at the end of the trace or a new trace has been loaded, the dispatcher checks if the trace contains a *Data Transformation* field (Figure 8). If so, the dispatcher brings the payload data into its Data Transform Engine (DTE in Figure 10), performs the transformation, and stores the data back to the output queue entry (Figure 8). Recall from Section III that these transformations are very simple. They involve changing between string, BSON, JSON, and similar formats. The engine needed is a simplified form of a (De)Ser accelerator [44], without the support for nested messages or custom data types. If other application domains or accelerators require other transformations, the engine will need to be revisited (e.g., as in DRX [81]).

Then, the dispatcher moves the contents of the output queue entry to the input queue of the accelerator identified by the PM in the trace. To perform this task, the dispatcher finds a free A-DMA engine and programs it to move the output queue entry to the input queue of the next accelerator. Once the engine confirms the transfer end, the dispatcher clears the output queue entry and terminates the operation. If needed for the transfer, the system correctly handles the logic of the *Memory Pointer* field in the queues and the *Overflow Pointer* associated with the destination input queue.

3. Interconnection Network, Memory Hierarchy, and Addressing. In the same way as the cores in a chiplet are connected in a mesh, the accelerators in a chiplet are also connected in a mesh (Figure 6). The different chiplets are connected with a high-bandwidth network. Like the cores, the accelerators access the cache-coherent, distributed LLC of the cores and, if they miss there, they access memory. Accelerator scratchpads and queues are directly addressed and, thus, not cacheable.

Cores and accelerators share the virtual address space. Each accelerator has a TLB that is also accessed by the input/output dispatchers. On a TLB miss, the IOMMU shared by the co-located accelerators loads the correct translation into the TLB. On a page fault or any other exception, the accelerator stops, the CPU is interrupted, and OS handles the exception.

4. Programming AccelFlow. AccelFlow proposes a new programming model and API. The software annotates the code of a service with which sections should be executed on which accelerator. The software also creates one or more traces, which are graphs of accelerator invocations. A trace does not have glue code between the accelerators invoked in sequence—except for branch conditions and data format transformations. Branches are encoded as simple operations on one or multiple fields of an accelerator's output. Data transformations encode source and destination data format.

Currently, programmers do these operations, constructing traces either by using predefined templates (e.g., those in Table II) or by explicitly building new traces through the API. The API specifies three aspects: accelerator call graph, branches, and data format changes. In future work, we will explore automating trace generation via compiler and runtime infrastructures. Our simulator takes the instrumented code and the traces, and models the execution in the simulated AccelFlow architecture.

A trace operates within a single service. It cannot span multiple services, to ensure independent scheduling and avoid cross-service dependencies. Within a service, AccelFlow allows traces to be triggered by network messages such as an RPC request, or by a core. Traces remain confined to user-space code. We do not support acceleration of kernel operations.

AccelFlow API. The AccelFlow API allows programmers to construct new traces. It has three components:

- *seq(*accels)*: Defines a linear chain of accelerators.
- *branch(condition-op, on-true, on-false)*: Adds a conditional control flow based on outputs of the previous accelerator.
- *trans(src, dst)*: Transforms the format of the data from one representation to another.

Trace Construction Example. Suppose we want to create the trace in Figure 4a. The sequence of accelerators to use is TCP, Decr, RPC, and Dser, then a branch condition that may call Dcmp or not, and finally LdB. Further, if Dcmp is to be invoked, a change in data format from JSON to string is required before Dcmp is invoked.

```
from AFlow import Trace, seq, branch, transform
trace = Trace() # Define trace
pipeline = seq( # Compose trace
  "TCP", "Decr", "RPC", "Dser",
  branch(condition_op="out['compressed'] == 1",
        on_true=seq(trans("JSON", "str"), "Dcmp"),
        on_false=None),
  "LdB")
trace.build(pipeline) # Attach pipeline to trace
trace.register(name="func_req") # Register trace
```

Listing 1: Constructing the trace in Figure 4a.

TABLE III: Architectural parameters used in evaluation.

Processor Parameters	
Processor	36 6-issue cores, 2.4GHz, 352/200 entry ROB/LSQ
L1 D-TLB	128 entries, 4-way, 2 cyc. round trip (RT)
L1 I-TLB	128 entries, 4-way, 2 cycles RT
L2 TLB	2048 entries, 8-way, 12 cycles RT
L1 D-Cache	48KB, 12-way, 5 cyc. RT, 64B line,16 MSHRs
L1 I-Cache	32KB, 8-way, 5 cyc. RT, 64B line, 16 MSHRs
L2 Cache	512KB, 8-way, 13 cycles RT, 32 MSHRs
LLC Slice	2MB, 16-way, 36 cycles RT, 32 MSHRs
AccelFlow Parameters	
Accel. Queues	64 entries in input queue and 64 in output queue
A-DMA Engines	10
PEs/Accelerator	8
Scratchpad	64 KB per PE in each accelerator
Queue to Scrtchpd	10 ns latency and 100 GB/s BW for 1KB msgs
Notification	Avg 80 cycles for an accelerator to notify a core
Intra-Chiplet Net	2D mesh, 3 cycles/hop, 16B links
Inter-Chiplet Net	Fully connected, 60 cycles [45], 1Gb/s/link
Main-memory Parameters	
Size; Rate	128GB; DDR
Controllers	4 mem. controllers; 4 channels per mem. cntr.
Mem. BW	102.4GB/s per memory controller

Listing 1 shows the code that constructs the trace. It is mostly self-explanatory. The branch node triggers the Dser's output dispatcher to compute the branch condition (i.e., check if the *out['compressed']* bit is set), and to invoke the correct subpath. The data transformation node triggers the Dser's output dispatcher to transform the data from JSON to string.

To use this trace inside the code of a service, developers simply invoke the trace with the *run_trace* function. This is shown in Listing 2. This code is triggered by an incoming function request (through *conn.recv*). *run_trace* takes the previously registered trace name (*func_req*) and the input for the first accelerator in the chain (*inReq*). In addition, the developer provides the *cpu_fallback* routine that is invoked if any exception is triggered during the trace execution (e.g., an accelerator is overloaded).

```
while True:
    conn, addr = accept_connection()
    try:
        inReq = conn.recv()
        processed = run_trace("func_req", inReq)
        ...
    except TraceError:
        result = cpu_fallback(inReq)
    conn.send(response)
    conn.close()
```

Listing 2: Code to invoke a trace on reception of a request.

VI. Evaluation Methodology

Modeled Architectures. We model a server-class processor with 36 cores and 128GB of main memory. Cores and caches are modeled after the Intel Sunny Cove microarchitecture [13], [14], [83] present in IceLake server processors [32]. The processor has two chiplets: one with 36 cores and the LdB accelerator, and the other with our remaining 8 accelerators. Table III shows the architectural parameters.

We model nine accelerators proposed in the literature: F4T [7] for TCP, QTLS [28] for (De)Encryption, Cerebros [63] for RPC, ProtoAcc [44] for (De)Serialization, CDPU [45]

(De)Compression, and Intel DLB [33] for load balancing. Each accelerator has 8 PEs.

We evaluate five servers. *Non-acc* has no accelerators. The other four servers orchestrate the same set of nine accelerators differently. They are: (1) *CPU-Centric* (as in Section III, accelerators are orchestrated by cores); (2) the state-of-the-art *RELIEF* [26] design (as representative of accelerators being orchestrated by a hardware manager); (3) the state-of-the-art *Cohort* [82] design (which links pairs of accelerators that frequently go together, but otherwise relies on the cores to orchestrate the accelerators); and (4) *AccelFlow*.

Simulation Infrastructure. We evaluate the architectures with full-system simulations using QEMU [69] and SST [66]. QEMU captures both user-space and kernel-space instructions, memory accesses, and system calls. QEMU passes all the events to the SST Ariel core [77], which we modified for high accuracy. The resulting system models the architectures and performs cycle-level simulations. The simulation environment models the whole software stack: OS (Ubuntu 24.04, Linux 6.8.0-71), container runtime (Docker-compose [16]), and the application. Main memory is modeled with DRAM-Sim2 [67].

Our simulation methodology explicitly models the data movement to/from accelerator scratchpads, accelerator scratchpad initialization and wipeout, DMA latency, cache and TLB misses, page table walks using radix-based page tables, LLC accesses and snoops, and contention on the on-package network. The network model accounts for both latency and bandwidth constraints.

How We Model the Accelerators. An accelerator PE operates in three steps: the input dispatcher loads the inputs into the scratchpad, the PE performs its computation C, and the PE saves the results into the output queue. Once the first step is complete, the PE performs the computation C in isolation, without system interaction. Hence, for a given input data size, the execution is highly deterministic.

Keeping this in mind, since the RTL-level design of the accelerators is unavailable to us, we estimate the time taken by C indirectly, as follows. The literature provides, for each accelerator and input data set, the speedup S that the accelerator delivers in its computation of C relative to a CPU. Then, in our simulations, we use these speedups. Specifically, in our simulations, we model a CPU and measure how many cycles it takes to execute computation C. Then, we assume that the accelerator takes C/S time to perform the same computation. While we use speedups S that depend on the input data size, on average across all input data sizes, the speedups S in the literature are: 3.5 for F4T, 6.6 for QTLS, 20.5 for Cerebros, 3.8 for ProtoAcc, 4.1 and 15.2 for CDPU decompression and compression, and 8.1 for LdB.

In our context, where the accelerator execution time is highly deterministic, not simulating the hardware inside the accelerators and, instead, using a specified number of cycles from the literature (which take into account the input data sizes) is a sound abstraction. However, to validate our approach, for some experiments, we also use gem5 models of accelerators [25] and observe similar results (Section VII-A4).

Applications. We run 8 SocialNetwork services from DeathStarBench [19]. To model a realistic environment, we take Alibaba's production-level open source traces [54] and pick 8 representative services from them that have similar size and call structure as the 8 SocialNetwork services. Then, we use the real-world invocation rates of those Alibaba services in our evaluation. The average load per service is 13.4K requests per second (RPS). We also run experiments with different loads. In this case, we use Poisson distributions for the request inter-arrival time. We use average loads of 5K, 10K, and 15K RPS. These experiments also include the HotelReservation and MediaServices from DeathStarBench.

Area Overhead of AccelFlow. We compute the hardware area via McPAT [51]. We use the 32nm technology available with the tool, and then scale to 7nm [73]. The total area of our baseline processor without accelerators is $122.3mm^2$: $83.1mm^2$ are the cores and their private caches, $38.2mm^2$ the LLC, and $1.0mm^2$ the network. The literature only provides the areas of the (De)Serialization [44] and (De)Compression [45] ASIC accelerators. Using their data, the area of the Ser, Dser, Cmp, and Dcmp accelerators with 8 PEs and 8 scratchpads each is $0.6mm^2$, $0.9mm^2$, $9.1mm^2$, and $5.2mm^2$, respectively. Based on their similar functionality, we estimate TCP and (De)Encr to have similar area as Cmp, and RPC and LdB to have similar area as Dser. These nine accelerators, with 8 PEs and 8 scratchpads each, take $44.9mm^2$.

Each accelerator has 64-entry input and output queues, and input/output dispatchers. Each entry in the queues is 2.1KB. For the input/output dispatchers, we conservatively estimate that each dispatcher has the same area as Dser. These queues and dispatchers for all accelerators take $3.4mm^2$, while the 10 A-DMA engines take $1.3mm^2$ [6] and the accelerator network takes $0.4mm^2$. The rest of the hardware in the accelerator chiplet takes negligible area.

Overall, of the total processor area, the combination of accelerators, queues, dispatchers, and accelerator network takes 29.0%, while the accelerators take 26.1% of the total area. Hence, AccelFlow's area overhead is at most 2.9% of the SoC.

VII. Evaluation

In this section, we evaluate the performance improvements with AccelFlow, characterize AccelFlow's operation, and perform sensitivity analyses.

A. Performance Improvements with AccelFlow

1. End-to-End Tail and Average Latency. The bars in Figure 11 show the P99 tail latency of services in the five architectures considered. The stars are the average latency. In all services, *AccelFlow* has the shortest tail, followed by *RELIEF* or *Cohort*, then *CPU-Centric*, and then *Non-acc*. On average, *AccelFlow* reduces the tail latency over *Non-acc*, *CPU-Centric*, *RELIEF*, and *Cohort* by 90.7%, 81.2%, 68.8% and 70.1%, respectively. These are large reductions. *AccelFlow* attains greater reductions for services that most frequently invoke accelerators (e.g., *CPost*), or that have frequent

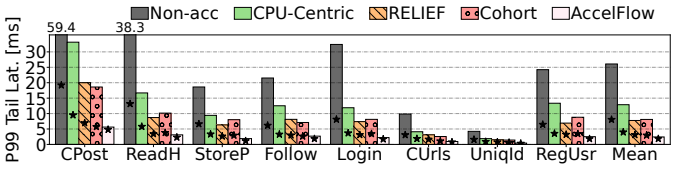

Fig. 11: P99 tail latency of services in different architectures. Black stars indicate the *average* latency of services.

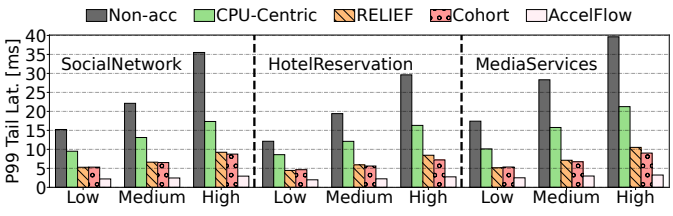

Fig. 12: P99 tail latency of services under different system loads in the five architectures considered.

branches in the trace (e.g., *Login*). In these, fast accelerator communication and branch resolution help.

The longer tail latency of RELIEF over AccelFlow does not come from long communication latencies between accelerators and the RELIEF orchestrator (the latter is in the same chiplet as the accelerators). RELIEF's limitation is that the orchestrator becomes a bottleneck. Let us ignore the execution time of an accelerator and the communication time between accelerator and orchestrator. Every time an accelerator finishes, the time for the orchestrator to get interrupted plus to process the information is $\approx$1.5μs [26]. A Cpost request uses 87 accelerators (some in parallel) as we will see in Section VII-B. Assume a medium load of 10K RPS. The time the RELIEF orchestrator is busy for the 10K requests that arrive in 1 second is 1.3 seconds. This causes the longer latency of RELIEF.

Cohort and RELIEF have similar tail latency. Cohort helps reduce its tail latency by allowing some form of static chaining between a few accelerators (Section VI). This chaining can be exploited without involving the CPU and, by reducing centralized contention, helps reduce the tail latency in Cohort.

The average latency follows the same trends as the tail, although the impact of AccelFlow is smaller. *AccelFlow* reduces the average latency over *Non-acc*, *CPU-Centric*, *RELIEF*, and *Cohort* by 77.2%, 53.9%, 40.7%, and 37.9%, respectively.

Figure 12 shows the P99 tail latency of DeathStarBench applications with various system loads—rather than according to the real-world traces. We compare the architectures for 3 loads: Low (5K RPS), Medium (10K RPS), and High (15K RPS). *AccelFlow* significantly reduces the tail latency across all loads. However, it is more effective at higher loads because it relieves contention more effectively. For example, on average, it reduces the tail latency over *RELIEF* by 55.1%, 60.9%, and 68.3% for 5K, 10K, and 15K RPS, respectively.

2. End-to-End Tail Latency Breakdown. Figure 13 shows the contributions of the main techniques in *AccelFlow* that reduce tail latency with real-world production traces. We apply these techniques one by one. We start with *RELIEF*, where all 8 PEs of all 9 accelerator types share a single centralized queue.

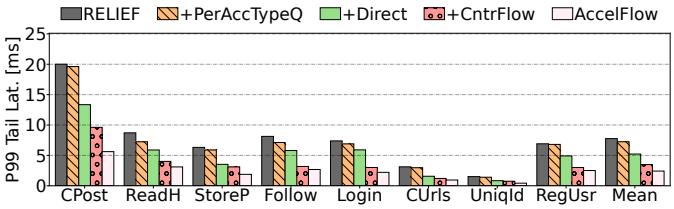

Fig. 13: P99 tail latency of services with the successive addition of *AccelFlow* techniques.

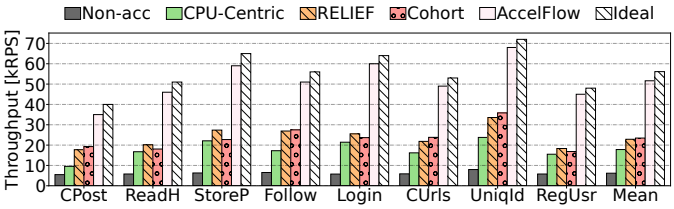

Fig. 14: Maximum throughput of the services in the five architectures plus an *Ideal* one.

With *PerAccTypeQ*, we distribute the queue so that there is a queue for each accelerator type. *Direct* additionally uses *traces* and supports direct data transfer between sequences of accelerators, eliminating the need for hardware manager intervention. *CntrFlow* additionally upgrades output dispatchers to resolve branches in the trace, eliminating hardware manager fallbacks. Finally, *AccelFlow* additionally upgrades dispatchers to perform data format transformations and handle large input data that does not fit in an input queue entry without involving the hardware manager.

All these techniques are effective. A separate queue per accelerator type (PerAccTypeQ) reduces contention and improves load balance. Direct accelerator-to-accelerator communication is effective, especially for CPost, which has long accelerator sequences. Having dispatchers that resolve branches in traces is beneficial, especially in services with frequent dynamic control flow such as Login. Finally, processing large payloads and performing data transformations is also effective. Overall, applying these techniques cumulatively reduces the average tail latency by 6.8%, 32.7%, 55.1%, and 68.7%.

3. Throughput Improvement with AccelFlow. Figure 14 shows the maximum throughput (i.e., the maximum load without violating the SLO) that the architectures attain. We define the SLO to be 5$\times$ the service execution time on an unloaded system [15], [58]. The figure also shows an *Ideal* system that allows the accelerators to communicate directly without incurring the overheads of branch resolution or data transformations. On average, *AccelFlow* improves throughput by 8.3$\times$ over *Non-acc* and by 2.2$\times$ over *RELIEF*. Further, *AccelFlow* is within 8.0% of the throughput of *Ideal*.

To further improve the performance, AccelFlow can use advanced scheduling policies instead of FIFO. We re-evaluate AccelFlow with a policy that prioritizes those requests that are closer to their deadline (Section IV-C). It can be shown that, with such a policy, AccelFlow improves the throughput by an additional 1.6$\times$.

4. Evaluation with a Gem5 Model of Accelerators. To

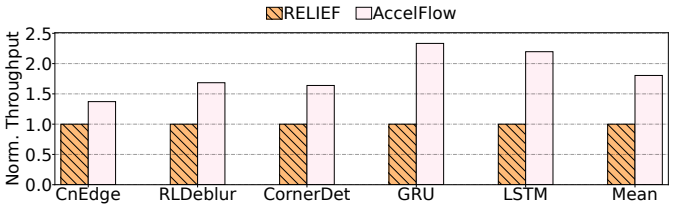

Fig. 15: Maximum throughput of image processing and RNN applications with *RELIEF* and *AccelFlow*.

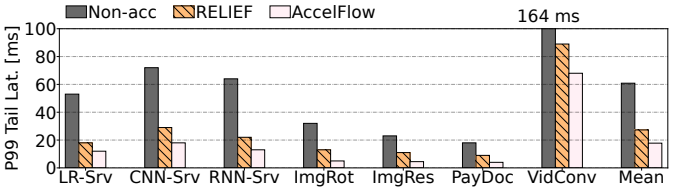

Fig. 16: P99 tail latency of serverless functions with Non-acc, RELIEF, and AccelFlow systems.

further validate and generalize AccelFlow, we re-evaluate RELIEF and AccelFlow using the gem5-based full-system simulator released with the RELIEF work. Their infrastructure includes 7 coarse-grain accelerators modeled in gem5 that are designed for image processing and RNNs. Hence, we run the image processing and RNN-based applications of the RELIEF benchmark suite, with the same configurations as in RELIEF's artifact. Figure 15 shows the maximum throughput of the applications in RELIEF and AccelFlow. We see that AccelFlow consistently achieves higher maximum throughput. On average, it improves throughput by 1.8× over RELIEF.

5. Evaluation with Serverless Functions. Beyond microservice environments, *AccelFlow* is also applicable to the emerging Function-as-a-Service (FaaS) or serverless computing paradigm. Serverless functions share many key characteristics with services: short execution times, bursty invocation patterns, and substantial datacenter tax overheads (e.g., encryption and serialization). To evaluate *AccelFlow* in this context, we use the open-source serverless benchmark suite *FunctionBench* [46], which includes workloads such as ML model serving, and image, video, and document processing.

We run these functions using production traces from Microsoft Azure [87], with all functions colocated on the same server. Figure 16 presents the per-function P99 tail latency across the *Non-acc*, *RELIEF*, and *AccelFlow* systems. *AccelFlow* substantially reduces the tail latency of serverless workloads, particularly for short-running functions such as *ImgRot*. On average, *AccelFlow* reduces the P99 tail latency of serverless functions by 37% compared to *RELIEF*.

B. Characterizing AccelFlow Operation

1. Components of the Execution Time. Figure 17 breaks down the execution time of a service in AccelFlow. It shows the time spent running on the CPU, running on the accelerators, executing the orchestration logic (dispatchers), and communicating between compute engines (i.e., accelerators and CPUs). The experiment runs on an unloaded system, with

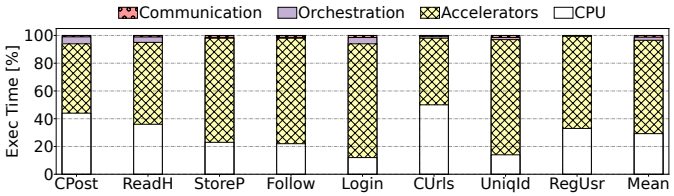

Fig. 17: Breakdown of a service execution time.

one request at a time, to avoid the effects of contention. We see that the time spent executing on accelerators dominates the total execution time, while the time spent on the orchestration logic is on average only 2.2%. This is in contrast to RELIEF, which can be shown to spend, on average, about 10% of the time on orchestration overheads.

2. Glue Instructions. The output dispatcher of an accelerator follows the flowchart of Figure 8. Typically, it finds no branch, end of trace, or data transformation. In this case, it executes about 15 RISC-like instructions. If it finds a branch, it resolves it and moves the Position Mark (Figure 8). The possible branch conditions are: Compressed?, Exception?, Hit?, and Found?. Resolving them involves checking a field in the output queue entry. On average, processing a branch adds the equivalent of 7 additional RISC instructions.

If an end of trace is found, the dispatcher either reads an ATM location and moves the trace in it to an accelerator's queue, or invokes a DMA to deposit the output data to memory, informs a CPU core, and clears the output queue entry. These operations take 12 to 20 RISC instructions.

If a data format transformation field is found, the dispatcher loads the source data in bulk, invokes the data transformation engine (Figure 10), and stores the data back to the output queue entry in bulk. The dispatcher executes 12 RISC instructions for 2KB payloads. Overall, in the worst case, an output dispatcher executes about 50 RISC instructions. In our services, the average number of instructions per output dispatcher operation is 18.

3. Characterizing Traces. We analyze the execution of each service in our benchmark suite and determine which traces of Table II it uses, in what sequence, and how many accelerators it invokes. Table IV shows the most common execution path and the total number of accelerators used per service invocation. For example, the CPost service executes trace T1, then goes to the CPU, then executes 4 parallel invocations of traces T9 and T10, returns to the CPU, then executes 3 parallel invocations of traces T9 and T10, returns to the CPU, and finally executes trace T2. Overall, services use 2-16 traces, and 9-87 accelerators per service invocation.

4. Accelerator Utilization. When operating at the system's peak throughput without violating SLOs, the accelerator utilizations are: TCP 92%, (De)Encr 82%, RPC 68%, (De)Ser 73%, (De)Cmp 38%, and LdB 71%. We see that only (De)Cmp has low utilization, as services invoke it less frequently. To improve utilization, one could reuse the accelerators for other tasks when they are idle. Currently, this is not possible because each accelerator is highly optimized for a single task.

TABLE IV: Most-common execution path per service and total number of accelerators used per service invocation.

Service	Most Common Execution Path	#
CPost	T1-CPU-4x(T9-T10)-CPU-3x(T9-T10)-CPU-T2	87
ReadH	T1-CPU-T4-T5-CPU-T9-T10-CPU-T3	28
StoreP	T1-CPU-T8-T7-CPU-T2	18
Follow	T1-CPU-3x(T8-T7)-CPU-T2	30
Login	T1-CPU-T4-T5-T6-T7-CPU-T2	29
CUrls	T1-CPU-T8-T7-CPU-T3	19
UniqId	T1-CPU-T2	9
RegUsr	T1-CPU-T8-T7-CPU-T9-T10-CPU-T2	25

Exploring programmable and more flexible accelerator designs is a direction for future work.

5. Power, Energy, and Memory. Using McPAT [51] as in Section VI, we compute the maximum power and the per-access energy of the AccelFlow structures in Table III: input/output queues, dispatchers, DMA transfers, on-package network, ATM, and other structures. For the accelerators themselves, we use the power and energy data from the literature. Input and output dispatchers are modeled as Dser engines. Based on our computations, the overall maximum power of the accelerators and of the AccelFlow orchestration structures is 12.5W and 5.0W, respectively. This is 3.1% and 1.2% of the maximum power of the server.

When running the DeathStarBench services with Alibaba's production invocation rates for 400K requests, we find that a server with AccelFlow reduces the energy consumption over one without accelerators (*Non-acc*) by 74%. In addition, a server with AccelFlow improves the performance per Watt by 7.2× over *Non-acc* and by 2.1× over RELIEF.

Compared to the other server designs with accelerators, AccelFlow adds 2.4MB of extra memory per server due to the input/output queues in the accelerators.

6. Frequency of High-Overhead Events. We measure how frequently AccelFlow encounters high-overhead events. These events are CPU fallbacks and TLB misses. The former can be due to a full overflow area, exceptions (including page faults), and TCP timeouts. The overflow area becoming full is infrequent, accounting for just 1.4% of all accelerator invocations on average, and up to 5.9% at peak load. Page faults are rare: 0.13 per million instructions. TCP accelerator input queue timeouts appear at a rate of 3.2 per million requests, primarily under bursty traffic conditions. Data and instruction L1 TLB misses occur at rates of 3.4 MPKI and 0.6 MPKI, respectively.

C. Sensitivity Analyses

We measure how different design choices for *AccelFlow* impact the end-to-end performance of services.

1. Processor Organization into Chiplets. Our processor has a core chiplet and an accelerator chiplet. One could design the processor with a single chiplet to reduce communication overheads, or with more chiplets to add flexibility. Figure 18 shows the P99 tail latency of services for different numbers of chiplets: *1-chiplet*; *2-chiplets* (the base design); *3-chiplets* (TCP and (De)Encr in one chiplet; RPC, (De)Ser, and (De)Cmp in another); *4-chiplets* (TCP and (De)Encr in one chiplet; RPC and (De)Ser in another; (De)Cmp in another); and *6-chiplets* (TCP, (De)Encr, RPC, (De)Ser, and (De)Cmp in separate chiplets). As we separate the accelerators into more chiplets, the inter-accelerator communication is more expensive. As a result, the tail latency of requests increases. The impact is significant: going from 2 to 6 chiplets increases the tail latency by 14% on average.

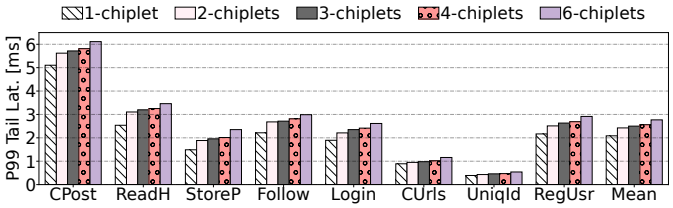

Fig. 18: P99 tail latency with different organizations of *AccelFlow* into chiplets.

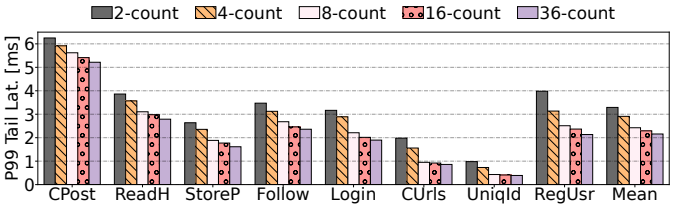

Fig. 19: P99 tail latency of services with different number of PEs per accelerator in *AccelFlow*.

2. Inter-chiplet Latency. Our default inter-chiplet latency is 60 cycles [45]. We also evaluate the tail latency of services for *AccelFlow* with different numbers of chiplets as we vary the inter-chiplet communication latency from 20 to 100 cycles. Inter-chiplet latency becomes more important as we increase the number of chiplets. It can be shown that, as we go from 60 to 100 cycles in 6-chiplet systems, the average tail latency increases by 45%.

3. PE Count. Figure 19 shows the P99 tail latency of services with different numbers of PEs per accelerator in *AccelFlow*. We said that, with 8 PEs per accelerator, on average, only 1.4% of all requests fall back to the CPU due to a full overflow area. When accelerators are provisioned with fewer PEs, fallback becomes increasingly common—particularly for accelerators with longer execution times, e.g., Encr and Cmp, and those that are invoked most frequently, e.g., TCP. With only 4 or 2 PEs, 16% and 39% of Encr requests, respectively, are denied accelerator access and instead fall back to CPU execution.

The fallback due to few PEs per accelerator degrades overall performance. As shown in Figure 19 the tail latency of services increases with fewer PEs per accelerator. Consider 4 and 2 PEs per accelerator. Compared to 8 PEs per accelerator, the tail latency increases by an average of 20.0% and 35.7%, respectively. Further, it can be shown that the system misses 8.2% of microservice request deadlines with 4 PEs and 21.7% with 2 PEs. These violations stem from both the increased latency of CPU execution and the system-wide contention introduced by fallback activity. Furthermore, the average throughput relative to that of 8 PEs drops by 11% and 25% with 4 and 2 PEs.

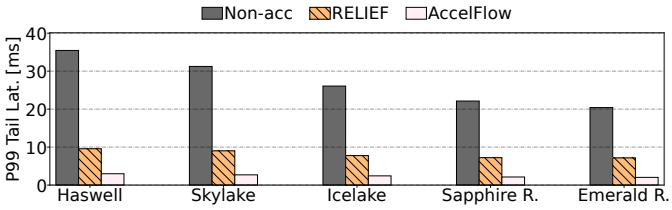

Fig. 20: P99 tail latency of services in Non-acc, RELIEF, and AccelFlow with different processor architectures.

4. Processor Generations. Newer processor generations can potentially enhance service performance thanks to improved single-thread capabilities—such as wider issue widths and larger caches, ROBs, LSQs, and TLBs. To evaluate this, we model several Intel processor generations, namely Haswell, Skylake, Ice Lake, Sapphire Rapids, and Emerald Rapids. Figure 20 shows the P99 tail latency for the *Non-acc*, *RELIEF*, and *AccelFlow* systems across these architectures (using the same experimental setup as in Figure 11).

While newer processors indeed improve the performance of the main service logic, they offer less benefit to datacenter tax operations. As a result, the relative cost of datacenter tax and accelerator orchestration becomes even more pronounced. Consequently, the relative performance advantage of *AccelFlow* grows with newer CPU architectures. For instance, while *AccelFlow* reduces the P99 tail latency over *RELIEF* by 68.8% on an Ice Lake–like CPU, it achieves a 71.7% reduction on an Emerald Rapids–like CPU.

5. Accelerator Speedups. We evaluate how the speedups attained by accelerators affect overall system performance. Starting with each accelerator delivering the speedup reported in the literature, we then multiply their attained speedup by 0.25×, 0.5×, 2×, and 4×. We then compare AccelFlow against RELIEF across these configurations. We see that, as accelerators deliver higher speedups, orchestration becomes more important, and AccelFlow becomes more attractive. At baseline speedups, AccelFlow outperforms RELIEF by 2.2× (Figure 14). With 4× and 0.25× the original speedups, AccelFlow gains are 3.9× and 1.4×, respectively.

VIII. Related Work

Datacenter Tax Profiling. Hyperscalers studied how cycles are spent in modern data centers [9], [20], [41], [42], [64], [68], [70], [71]. They observed that datacenter tax consumes a large fraction of cycles. Their profiling results motivated the need for hardware accelerators, potentially chained together into a *sea of accelerators* [20]. Their profiling measurements were done at a relatively high level, aggregating the results across many systems. There is not enough information to get insights into how to orchestrate multiple-accelerator execution. They focus on data analysis and not on system design.

Accelerators for Microservices. Many works explored the design of accelerators for individual sources of datacenter tax [1], [7], [28], [30], [33], [34], [38], [39], [43]–[45], [50], [63], [84]. These schemes efficiently reduce individual overheads, such as TCP [7] or RPC [30], [50], [63] processing. In contrast, AccelFlow is a scheme to efficiently *orchestrate* multiple accelerators. Researchers also proposed accelerators for application logic [52], [53], [85], such as key-value stores [52] or databases [85]. Any accelerator with the standard interface can be seamlessly integrated into AccelFlow. Finally, researchers proposed specialized CPU architectures for microservices [56], [57], [74], [89]. AccelFlow can be implemented on top of different processor architectures.

Accelerator Orchestration. Researchers proposed scheduling schemes for processors equipped with various accelerators [2], [5], [11], [12], [26], [27], [47], [60], [76], [82]. Many rely on a CPU core [27] or centralized hardware manager [12], [26] to orchestrate the accelerators. Cohort [82] uses a CPU-based acceleration framework based on shared-memory software queues to link a few accelerators. VIP [60] chains several accelerators so they appear to the software as a single pipelined device. VIP is applied to coarse-grained video-processing apps that always use the same accelerator chain; it cannot support the dynamic behavior of microservices. AuRORA [47] proposes a coarse-grained scheme to virtualize accelerators for multi-tenant DNN workloads. AuRORA is decentralized, but it does not orchestrate accelerator chains.

IX. Conclusion and Future Work

An ensemble of on-package accelerators, each targeting a different source of datacenter tax, has the potential to significantly improve the performance of microservice environments. However, realizing these benefits requires an accelerator orchestration framework that can keep up with the fine-grained, highly dynamic nature of microservices. Our characterization shows that such a framework must be both flexible and nimble.

To address this challenge, this paper presented AccelFlow, the first accelerator orchestration framework designed for microservices. In AccelFlow, the software constructs traces that encode sequences of accelerators and, optionally, branch conditions. Once a trace is enqueued, the accelerators in the trace execute in sequence without further CPU involvement, resolving branches on the fly and forwarding data directly among themselves. Compared to state-of-the-art orchestration schemes, AccelFlow on average reduces P99 tail latency by 70%, reduces average latency by 38%, and increases throughput by 120%.

AccelFlow's trace abstraction and decentralized orchestration open up new research directions in automated trace synthesis and optimization, QoS- and SLO-aware accelerator scheduling, and adaptive control-flow decisions based on real-time system load and microservice behavior. More broadly, it provides a concrete substrate for co-designing microservice runtimes, compilers, and future accelerators around programmable, fine-grained accelerator ensembles.

Acknowledgments

This work was supported in part by NSF under grants CCF 2107470 and CCF 2316233; and by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

REFERENCES

[1] B. Abali, B. Blaner, J. Reilly, M. Klein, A. Mishra, C. B. Agricola, B. Sendir, A. Buyuktosunoglu, C. Jacobi, W. J. Starke, H. Myneni, and C. Wang, "Data Compression Accelerator on IBM POWER9 and z15 Processors : Industrial Product," in *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA '20)*, 2020.

[2] A. Amarnath, S. Pal, H. T. Kassa, A. Vega, A. Buyuktosunoglu, H. Franke, J.-D. Wellman, R. Dreslinski, and P. Bose, "Heterogeneity-Aware Scheduling on SoCs for Autonomous Vehicles," *IEEE Computer Architecture Letters*, 2021.

[3] Apache, "Thrift," https://thrift.apache.org/, 2024.

[4] ARM, "Learn the Architecture - SMMU Software Guide," https://developer.arm.com/documentation/109242/0100/Operation-of-an-SMMU/Address-Translation-Services, 2023.

[5] S. Baskaran, M. T. Kandemir, and J. Sampson, "An architecture interface and offload model for low-overhead, near-data, distributed accelerators," in *Proceedings of the 55th IEEE/ACM International Symposium on Microarchitecture (MICRO '22)*, 2022.

[6] T. Benz, M. Rogenmoser, P. Scheffler, S. Riedel, A. Ottaviano, A. Kurth, T. Hoefler, and L. Benini, "A High-Performance, Energy-Efficient Modular DMA Engine Architecture," *IEEE Transactions on Computers*, 2024.

[7] J. Boo, Y. Chung, E. Baek, S. Na, C. Kim, and J. Kim, "F4T: A Fast and Flexible FPGA-based Full-stack TCP Acceleration Framework," in *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23)*, 2023.

[8] A. Branover, D. Foley, and M. Steinman, "AMD Fusion APU: Llano," *IEEE Micro*, vol. 32, no. 2, pp. 28–37, 2012.

[9] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger, "A cloud-scale acceleration architecture," in *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '16)*, 2016.

[10] M. Chabbi and M. K. Ramanathan, "A Study of Real-World Data Races in Golang," in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '22)*, 2022.

[11] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, and G. Reinman, "CHARM: A composable heterogeneous accelerator-rich microprocessor," in *Proceedings of the 2012 ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED '12)*, 2012.

[12] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, and G. Reinman, "Architecture Support for Domain-Specific Accelerator-Rich CMPs," *ACM Transactions on Embedded Computing Systems*, 2014.

[13] I. Cutress, "Examinining Intel's Ice Lake Processors: Taking a Bite of the Sunny Cove Microarchitecture," https://www.anandtech.com/show/14514/examining-intels-ice-lake-microarchitecture-and-sunny-cove/3, 2019.

[14] I. Cutress, "The Ice Lake Benchmark Preview: Inside Intel's 10nm," https://www.anandtech.com/show/14664/testing-intel-ice-lake-10nm/2, 2019.

[15] C. Delimitrou and C. Kozyrakis, "Amdahl's Law for Tail Latency," *Communications of the ACM*, vol. 61, no. 8, jul 2018.

[16] Docker, "Docker Compose," https://docs.docker.com/compose/, 2024.

[17] Facebook, "Zstandard," https://facebook.github.io/zstd/, 2024.

[18] Y. Gan, Y. Qiu, L. Chen, J. Leng, and Y. Zhu, "Low-Latency Proactive Continuous Vision," in *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques (PACT '20)*, 2020.

[19] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou, "An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*, 2019.

[20] A. Gonzalez, A. Kolli, S. Khan, S. Liu, V. Dadu, S. Karandikar, J. Chang, K. Asanovic, and P. Ranganathan, "Profiling Hyperscale Big Data Processing," in *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23)*, 2023.

[21] Google, "Protocol Buffers," https://protobuf.dev/, 2024.

[22] Google, "Snappy, a fast compressor/decompressor," https://github.com/google/snappy, 2024.

[23] Google Cloud, "What is Microservices Architecture?" https://cloud.google.com/learn/what-is-microservices-architecture, 2024.

[24] gRPC, "An RPC library and framework," https://github.com/grpc/grpc, 2024.

[25] S. Gupta and S. Dwarkadas, "RELIEF: Open Source Artifact," https://github.com/Sacusa/gem5-SALAM/blob/0f057711e43b03cced9b0cebeca6a5643086485e/benchmarks/scheduler/sw/runtime.c#L1623, 2024.

[26] S. Gupta and S. Dwarkadas, "RELIEF: Relieving Memory Pressure In SoCs Via Data Movement-Aware Accelerator Scheduling," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA '24)*, 2024.

[27] M. Hill and V. J. Reddi, "Gables: A Roofline Model for Mobile SoCs," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA '19)*, 2019.

[28] X. Hu, C. Wei, J. Li, B. Will, P. Yu, L. Gong, and H. Guan, "QTLS: high-performance TLS asynchronous offload framework with Intel® QuickAssist technology," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming (PPoPP '19)*, 2019.

[29] D. Huye, Y. Shkuro, and R. R. Sambasivan, "Lifting the veil on Meta's microservice architecture: Analyses of topology and request workflows," in *Proceedings of the USENIX Annual Technical Conference (USENIX ATC '23)*, 2023.

[30] S. Ibanez, A. Mallery, S. Arslan, T. Jepsen, M. Shahbaz, C. Kim, and N. McKeown, "The nanoPU: A Nanosecond Network Stack for Datacenters," in *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI '21)*, 2021.

[31] Intel, "Intel 64 and IA-32 Architectures Software Developer's Manual," 2024. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf

[32] Intel, "Intel Xeon Platinum 8380 Processor," https://ark.intel.com/content/www/us/en/ark/products/212287/intel-xeon-platinum-8380-processor-60m-cache-2-30-ghz.html, 2024.

[33] Intel, "Intel® Dynamic Load Balancer," https://www.intel.com/content/www/us/en/download/686372/intel-dynamic-load-balancer.html, 2024.

[34] Intel, "Intel® QAT: Performance, Scale, and Efficiency," https://www.intel.com/content/www/us/en/architecture-and-technology/intel-quick-assist-technology-overview.html, 2024.

[35] Intel, "Intel® Xeon® Platinum 8360Y Processor," https://www.intel.com/content/www/us/en/products/sku/212459/intel-xeon-platinum-8360y-processor-54m-cache-2-40-ghz/specifications.html, 2024.

[36] Intel, "Introduction to Memory Bandwidth Allocation," https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-memory-bandwidth-allocation.html, 2024.

[37] Intel, "Products formerly SAPPHIRE RAPIDS," https://ark.intel.com/content/www/us/en/ark/products/codename/126212/products-formerly-sapphire-rapids.html, 2024.

[38] J. Jang, S. J. Jung, S. Jeong, J. Heo, H. Shin, T. J. Ham, and J. W. Lee, "A Specialized Architecture for Object Serialization with Applications to Big Data Analytics," in *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA '20)*, 2020.

[39] K. Jang, S. Han, S. Han, S. Moon, and K. Park, "SSLShader: Cheap SSL Acceleration with Commodity Processors," in *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI '11)*, 2011.

[40] E. Jeong, S. Wood, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park, "mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems," in *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14)*, 2014.

[41] G. Jeong, B. Sharma, N. Terrell, A. Dhanotia, Z. Zhao, N. Agarwal, A. Kejariwal, and T. Krishna, "Characterization of Data Compression in Datacenters," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS '23)*, 2023.

[42] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, "Profiling a warehouse-scale computer," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA '15)*, 2015.

[43] S. Kanev, S. L. Xi, G.-Y. Wei, and D. Brooks, "Mallacc: Accelerating Memory Allocation," in *Proceedings of the Twenty-Second International*

Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17).

[44] S. Karandikar, C. Leary, C. Kennelly, J. Zhao, D. Parimi, B. Nikolic, K. Asanovic, and P. Ranganathan, "A Hardware Accelerator for Protocol Buffers," in *Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21)*, 2021.

[45] S. Karandikar, A. N. Udipi, J. Choi, J. Whangbo, J. Zhao, S. Kanev, E. Lim, J. Alakuijala, V. Madduri, Y. S. Shao, B. Nikolic, K. Asanovic, and P. Ranganathan, "CDPU: Co-designing Compression and Decompression Processing Units for Hyperscale Systems," in *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23)*, 2023.

[46] J. Kim and K. Lee, "FunctionBench: A Suite of Workloads for Serverless Cloud Function Service," in *Proceedings of the IEEE 12th International Conference on Cloud Computing (CLOUD '19)*, 2019.

[47] S. Kim, J. Zhao, K. Asanovic, B. Nikolic, and Y. S. Shao, "AuRORA: Virtualized Accelerator Orchestration for Multi-Tenant Workloads," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '23)*, 2023.

[48] R. Kuper, I. Jeong, Y. Yuan, R. Wang, N. Ranganathan, N. Rao, J. Hu, S. Kumar, P. Lantz, and N. S. Kim, "A Quantitative Analysis and Guidelines of Data Streaming Accelerator in Modern Intel Xeon Scalable Processors," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'24), Volume 2*, 2024.

[49] A. Kwatra, "Intel Labs' Contributions to Latest Intel® Xeon® Scalable Processor," https://community.intel.com/t5/Blogs/Tech-Innovation/Data-Center/Intel-Labs-Contributions-to-Latest-Intel-Xeon-Scalable-Processor/post/1441731, 2024.

[50] N. Lazarev, S. Xiang, N. Adit, Z. Zhang, and C. Delimitrou, "Dagger: efficient and fast RPCs in cloud microservices with near-memory reconfigurable NICs," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*, 2021.

[51] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '09)*, 2009.

[52] S. Li, H. Lim, V. W. Lee, J. H. Ahn, A. Kalia, M. Kaminsky, D. G. Andersen, O. Seongil, S. Lee, and P. Dubey, "Architecting to achieve a billion requests per second throughput on a single key-value store server platform," in *Proceedings of the ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA '15)*, 2015.

[53] K. Lim, D. Meisner, A. G. Saidi, P. Ranganathan, and T. F. Wenisch, "Thin servers with smart pipes: designing SoC accelerators for memcached," in *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*, 2013.

[54] S. Luo, H. Xu, C. Lu, K. Ye, G. Xu, L. Zhang, Y. Ding, J. He, and C. Xu, "Characterizing Microservice Dependency and Performance: Alibaba Trace Analysis," in *Proceedings of the ACM Symposium on Cloud Computing (SoCC '21)*, 2021.

[55] S. Luo, H. Xu, K. Ye, G. Xu, L. Zhang, G. Yang, and C. Xu, "The Power of Prediction: Microservice Auto Scaling via Workload Learning," in *Proceedings of the ACM Symposium on Cloud Computing (SoCC '22)*, 2022.

[56] A. Mirhosseini, H. Golestani, and T. F. Wenisch, "HyperPlane: A Scalable Low-Latency Notification Accelerator for Software Data Planes," in *Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '20)*, 2020.

[57] A. Mirhosseini, A. Sriraman, and T. F. Wenisch, "Enhancing Server Efficiency in the Face of Killer Microseconds," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA '19)*, 2019.

[58] A. Mirhosseini and T. Wenisch, "μSteal: A Theory-Backed Framework for Preemptive Work and Resource Stealing in Mixed-Criticality Microservices," in *Proceedings of the ACM International Conference on Supercomputing (ICS '21)*, 2021.

[59] MongoDB, "Explaining BSON with examples," https://www.mongodb.com/basics/bson, 2024.

[60] N. C. Nachiappan, H. Zhang, J. Ryoo, N. Soundararajan, A. Sivasubramaniam, M. T. Kandemir, R. Iyer, and C. R. Das, "VIP: Virtualizing IP chains on handheld platforms," in *Proceedings of the ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA '15)*, 2015.

[61] Old GigaOm, "The biggest thing Amazon got right: The platform," https://old.gigaom.com/2011/10/12/419-the-biggest-thing-amazon-got-right-the-platform/, 2011.

[62] OpenSSL, "Cryptography and SSL/TLS Toolkit," https://www.openssl.org/, 2024.

[63] A. Pourhabibi, M. Sutherland, A. Daglis, and B. Falsafi, "Cerebros: Evading the RPC Tax in Datacenters," in *Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21)*, 2021.

[64] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, "A reconfigurable fabric for accelerating large-scale datacenter services," in *Proceedings of the ACM/IEEE 41st International Symposium on Computer Architecture (ISCA '14)*, 2014.

[65] C. Richardson, "What are microservices?" https://microservices.io/, 2024.

[66] A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. D. Kersey, R. A. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. Cooper-Balis, and B. L. Jacob, "The structural simulation toolkit," *SIGMETRICS Performance Evaluation Reviews*, vol. 38, no. 4, 2011.

[67] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "DRAMSim2: A Cycle Accurate Memory System Simulator," *IEEE Computer Architecture Letters*, 2011.

[68] K. Seemakhupt, B. E. Stephens, S. Khan, S. Liu, H. Wassel, S. H. Yeganeh, A. C. Snoeren, A. Krishnamurthy, D. E. Culler, and H. M. Levy, "A Cloud-Scale Characterization of Remote Procedure Calls," in *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP '23)*, 2023.

[69] Software Freedom Conservancy, "QEMU: A generic and open source machine emulator and virtualizer," https://www.qemu.org/.

[70] A. Sriraman and A. Dhanotia, "Accelerometer: Understanding Acceleration Opportunities for Data Center Overheads at Hyperscale," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*, 2020.

[71] A. Sriraman, A. Dhanotia, and T. F. Wenisch, "SoftSKU: Optimizing Server Architectures for Microservice Diversity @Scale," in *Proceedings of the 2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA '19)*, 2019.

[72] A. Sriraman and T. F. Wenisch, "µSuite: A Benchmark Suite for Microservices," in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC '18)*, 2018.

[73] A. Stillmaker and B. Baas, "Scaling equations for the accurate prediction of CMOS device performance from 180nm to 7nm," *Integration the VLSI journal*, 2017.

[74] J. Stojkovic, C. Liu, M. Shahbaz, and J. Torrellas, "µManycore: A Cloud-Native CPU for Tail at Scale," in *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23)*, 2023.

[75] N. Sun and P. Bhattacharya, "Using the Cryptographic Accelerator of the UltraSPARC T1 Processor," March 2006. [Online]. Available: https://www.oracle.com/technetwork/server-storage/solaris/documentation/819-5782-150147.pdf

[76] C. Tan, M. Karunaratne, T. Mitra, and L.-S. Peh, "Stitch: Fusible Heterogeneous Accelerators Enmeshed with Many-Core Architecture for Wearables," in *Proceedings of the ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA '18)*, 2018.

[77] The Structural Simulation Toolkit, "ariel," https://sst-simulator.org/sst-docs/docs/elements/ariel/intro, 2024.

[78] Think Software, "Microservices Architecture of Twitter Service," https://thinksoftware.medium.com/design-twitter-microservices-architecture-of-twitter-service-996ddd68e1ca, 2021.

[79] Uber, "Introducing Domain-Oriented Microservice Architecture," https://www.uber.com/blog/microservice-architecture/, 2020.

[80] K. Varshneya, "Understanding design of microservices architecture at Netflix," https://www.techaheadcorp.com/blog/design-of-microservices-architecture-at-netflix/, 2021.

[81] S.-T. Wang, H. Xu, A. Mamandipoor, R. Mahapatra, B. H. Ahn, S. Ghodrati, K. Kailas, M. Alian, and H. Esmaeilzadeh, "Data Motion Acceleration: Chaining Cross-Domain Multi Accelerators," in *Proceedings*

of the IEEE International Symposium on High-Performance Computer Architecture (HPCA'24), 2024.

[82] T. Wei, N. Turtayeva, M. Orenes-Vera, O. Lonkar, and J. Balkind, "Cohort: Software-Oriented Acceleration for Heterogeneous SoCs," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '23)*, 2023.

[83] Wikipedia, "Sunny Cove," https://en.wikipedia.org/wiki/Sunny_Cove_(microarchitecture).

[84] A. Wolnikowski, S. Ibanez, J. Stone, C. Kim, R. Manohar, and R. Soulé, "Zerializer: towards zero-copy serialization," in *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '21)*, 2021.

[85] L. Wu, A. Lottarini, T. K. Paine, M. A. Kim, and K. A. Ross, "Q100: the architecture and design of a database processing unit," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*, 2014.

[86] Y. Yuan, J. Hu, R. Wang, N. Ranganathan, R. Kuper, I. Jeong, and N. S. Kim, "On-chip Accelerators in 4th Gen Intel Xeon Scalable Processors: Features, Performance, Use Cases, and Future!" ISCA'23 Tutorial, 2023.

[87] Y. Zhang, Goiri, G. I. Chaudhry, R. Fonseca, S. Elnikety, C. Delimitrou, and R. Bianchini, "Faster and Cheaper Serverless Computing on Harvested Resources," in *Proceedings of the International Symposium on Operating Systems Principles (SOSP '21)*, 2021.

[88] Z. Zhang, M. K. Ramanathan, P. Raj, A. Parwal, T. Sherwood, and M. Chabbi, "CRISP: Critical Path Analysis of Large-Scale Microservice Architectures," in *USENIX Annual Technical Conference (USENIX ATC '22)*, 2022.

[89] J. Zhao, I. Uwizeyimana, K. Ganesan, M. C. Jeffrey, and N. E. Jerger, "ALTOCUMULUS: Scalable Scheduling for Nanosecond-Scale Remote Procedure Calls," in *Proceedings of the 55th IEEE/ACM International Symposium on Microarchitecture (MICRO '22)*, 2022.

[90] X. Zhou, X. Peng, T. Xie, J. Sun, C. Xu, C. Ji, and W. Zhao, "Benchmarking Microservice Systems for Software Engineering Research," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings (ICSE '18)*, 2018.