UNIVERSITY OF CALIFORNIA SAN DIEGO

Operating System Scheduling for Emerging Hardware Accelerators

A thesis submitted in partial satisfaction of the
requirements for the degree
Master of Science

in

Computer Science

by

Abhishek Vijeev

Committee in charge:

Professor Amy Ousterhout, Chair
Professor George Porter
Professor Geoffrey Voelker

2023

University of California San Diego

2023

To Grandad, Grandmom, Mom and Sis

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

Foremost, dearest Grandad and Grandmom, I give you my deepest gratitude for financing my undergraduate education and for wholeheartedly supporting me when I expressed interest in pursuing my love for computer systems at UCSD - a career in Computer Science would have remained a dream without your love and support. Dearest Mom, thank you for being my pillar of support, for your undying love, optimism and encouragement during the most trying times, for always being available when I needed someone to talk to and for pushing me to be a better version of myself each day. Little Sis, thank you for being my best friend and for cheering me up on difficult days. I don't know where I'd be without you all - thank you :)

Professor Amy Ousterhout, thank you so much for being my advisor and for the opportunity to work on this fun project, which I hope has helped me understand computer systems a little better. Whenever a technical concept eluded me, you took the trouble to explain things in excruciating detail. You gave me constant feedback, helped me set tractable short-term goals and were a source of constant encouragement. Thank you so much for painstakingly reading through multiple drafts of this thesis - you went beyond the call of duty on many occasions to make sure that I received timely feedback. This thesis would not have been possible without your kindness, guidance, constant support and attention to detail. Lastly, you were very kind to overlook the numerous mistakes I made along the way - they have taught me valuable lessons. You have been instrumental in shaping my research outlook and in helping me work towards my goals at UCSD - I'm extremely fortunate and grateful to have had you as my advisor. Thank you! :)

Professor Geoff Voelker, thank you for serving on my committee, for giving me feedback on my work and most importantly, for teaching me graduate operating systems. Thank you so much for making time to answer all my questions, inside and outside the classroom - I truly enjoyed our discussions on OS design philosophy; to this day, I cannot fathom how deep your knowledge runs :) CSE 221 played a foundational role in helping me undertake this thesis - it will always remain close to my heart and I will cherish having been your student. Thank you.

Professor George Porter, thank you for serving on my committee, for giving me feedback on my work and for teaching me computer networking. Moreover, thank you so much for giving me the opportunity to serve as a Teaching Assistant - I cannot overstate how much it has helped me financially. Thank you.

Saarth Deshpande, thank you for porting the web server application used in Chapter 4.2.2's evaluation to Caladan - this was really helpful!

The MS Advisors - Emily, Dema and Nadyne, thank you for being available to answer questions about coursework/research and for making sure that I was on track to meeting my deadlines. Moreover, thank you so much for helping me navigate Fall'21.

I also express my sincere gratitude to UCSD CSE for allowing us to defer our admits during the Covid-19 pandemic - this gesture was truly magnanimous.

Finally, to the doctors who saved my life - I would not have made it to UCSD on time if it hadn't been for your dexterity and benevolence. I will always be indebted to you.

ABSTRACT OF THE THESIS

Operating System Scheduling for Emerging Hardware Accelerators

by

Abhishek Vijeev

Master of Science in Computer Science

University of California San Diego, 2023

Professor Amy Ousterhout, Chair

Hardware accelerators are becoming increasingly important in (1) meeting application performance demands and (2) mitigating datacenter software overheads. However, offloading computation to accelerators incurs fundamental overheads, which must be accounted for when measuring their benefits. In this thesis, we study the characteristics of a modern compression accelerator and show that only certain offload granularities yield performance gains. Further, we show that compressing large buffers ($\geq$ 8KB) with an off-chip accelerator yields latencies within 25% of the offload latency for an integrated compression accelerator that resides on the same chip as a SmartNIC's ARM cores, indicating the viability of off-chip acceleration. We use these insights to design and implement a microsecond-scale operating system scheduler that selectively offloads the parts of an application that benefit from hardware acceleration. Moreover, we design scheduling policies that decide whether hardware offload should be performed synchronously or asynchronously, depending on workload characteristics. Preliminary evaluation on synthetic applications closely modeled after real datacenter workloads shows that our system achieves up to 28% lower median latency and up to 3.8x higher overall throughput than a software-based approach by efficiently offloading compression to hardware accelerators.

Introduction

With the end of Moore's Law and Dennard Scaling [35], computer architects have embraced the need for specialized hardware accelerators to keep up with the rapidly increasing performance demands of application software, while improving energy efficiency. For instance, GPUs [20] have traditionally been used to accelerate highly parallel workloads including computer graphics and simulations. More recently, domain-specific accelerators such as TPUs for machine learning [43], virtualization hardware [4] and SmartNICs for networking [36] have improved application performance by over an order of magnitude, resulting in a renewed interest in hardware specialization.

Moreover, the past decade has witnessed a surge in warehouse-scale computing, as used by large internet companies for two key purposes: (1) to host the infrastructure that powers today's cloud and (2) to run client-facing production workloads. Warehouse-scale applications, which were traditionally implemented as monolithic software, are increasingly designed as microservices [1, 7, 26, 27] that require end hosts to perform a wide range of operations, such as decompression, deserialization and decryption, before being able to execute their core functionality. These operations, commonly referred to as the "datacenter tax", account for a massive 30% of all fleet-wide CPU cycles [46]. Fortunately, studies have shown that these overheads arise from a set of highly mature software modules, which are highly amenable to hardware acceleration. As a result, emerging hardware supports accelerators for commonly used operations such as compression, hashing and cryptography [18, 21], with accelerators for Remote Procedure Calls (RPCs) [41, 51, 61], serialization/deserialization [48] and memory allocation [47] around the corner.

Given the increasing prevalence of domain-specific accelerators, is it beneficial to offload all relevant computation to hardware? Offloading computation to hardware accelerators incurs overheads, which include (1) setting up the accelerator (by the host CPU) before an operation can be issued and (2) the latency of actually executing the operation on the accelerator, both of which depend on a number of factors [28]. In this thesis, we study the overheads of accelerator setup, as well as how two factors, namely (a) offload data size and (b) accelerator location, contribute to offload latency. To this extent, we first study the relationship between input buffer sizes and offload overheads for a compression accelerator found on Nvidia Bluefield 2 SmartNICs [21], and compare the results obtained with the cost of performing compression on host x86 cores using a state-of-the-art software compression library [16].

Moreover, recent trends towards integrated accelerators [19, 38, 41, 48] that are tightly coupled with CPU cores eliminate the overheads of traversing the PCIe bus. To better understand how on-chip (integrated) accelerators influence application performance, we also measure the overheads of offloading compression directly from the Bluefield's ARM cores to its on-chip accelerators.

Our experiments reveal that hardware offload to an on-chip compression accelerator from the Bluefield's ARM cores yields lower latencies than software compression for all kinds of input data and buffer sizes studied. In contrast, compressing small buffers (< 256 bytes) in software with x86 cores achieves lower latencies than hardware offload over PCIe. While PCIe overheads dominate at small offload granularities, we observe that these overheads are amortized by the high offload latency for large buffers, making off-chip acceleration an attractive alternative to running applications completely on a SmartNIC's wimpy cores, which have access to limited memory. In addition, the accelerator's setup time for both on-chip and off-chip offload remains lower than all other measured values. Consequently, applications whose requests aren't latency-critical can achieve higher throughput by always offloading computation to the accelerator.

Using these insights, we design and implement a microsecond-scale operating system scheduler that uses information about (1) request characteristics to determine when computation must be offloaded to hardware and (2) an application's performance goals to decide whether hardware offload should be performed synchronously or asynchronously. Our scheduler achieves low latency offload with the help of kernel bypass libraries [13] that provide direct access to hardware accelerators. Moreover, our implementation leverages user-level threading for nanosecond-scale context switches between application threads, minimizing the impact of context-switching latency on the throughput of asynchronous offload.

Evaluation on synthetic applications whose compression request distributions mirror real datacenter workloads [67] reveals that efficiently offloading computation to hardware accelerators achieves 3.8x higher throughput and 23% lower median latency when compared to performing compression in software. To also understand performance on real applications, we measure throughput and per-request service latencies for a synthetic web-service application that compresses large segments of data before returning them to clients - our system achieves 2.14x higher throughput and 28% lower median latency in comparison to software compression. Interestingly, though our custom scheduling policies exploit the observation that small buffers do not benefit from hardware offload, the workloads we evaluate seldom compress small buffers, as a result of which, policies that always offload compression to an accelerator perform similarly in comparison to dynamic policies that adaptively choose between software compression and hardware offload.

The rest of this thesis is organized as follows. Chapter 1 provides the necessary background on datacenter software overheads, hardware accelerators and hardware offload latency. Chapter 2 studies the performance of a compression accelerator found on Nvidia Bluefield SmartNICs. Chapter 3 outlines our system's design goals, describes how our proposed design achieves these goals and also provides details about our implementation. Chapter 4 evaluates the performance of applications that use our system to meet their performance goals and also studies the impact of parameters on the latency and throughput of asynchronous offload. Chapter 5 discusses related work in this area and puts this thesis in context. Chapter 6 highlights extensions to our system that we're currently working on. Chapter 7 concludes this thesis.

Chapter 1

Background

1.1 Datacenter Tax

Web search, social networks, media streaming and retail form a major fraction of today's datacenter applications. While such applications were traditionally implemented and deployed as monolithic software, they are increasingly being deployed as microservices [1, 7, 26, 27], wherein a complex application is decomposed into smaller distributed applications, each of which is tasked with fulfilling a specific subset of the application's overall functionality. The move to a microservice-based architecture has been influenced by a number of factors, including the ability to quickly innovate and scale compute resources [71]. However, decomposing applications into distributed microservices engenders the need for message-based communication over the network, which is most commonly achieved with Remote Procedure Calls (RPCs) [6, 8]. Consequently, each microservice must perform a range of different operations, including decompression, deserialization and decryption, before being able to execute its core functionality. Each of these operations must be performed for every request processed by a microservice, leading to fundamental performance/operational overheads, dubbed the "datacenter tax" [46], that account for approximately 30% of all fleet-wide CPU cycles. As successive generations of server hardware yield diminishing performance returns, minor improvements in CPU efficiency can save millions of dollars [48, 70], due to which, achieving high CPU efficiency is paramount. A trivial solution to improve CPU efficiency would be to reduce the datacenter tax by reverting to a monolithic architecture. While recent results have shown that such a move could improve scalability and lower costs for specific applications [2], in general, the microservice architecture remains popular today because its benefits outweigh its shortcomings. As a result, a widespread shift to monolithic applications is unlikely in the near future. However, is it possible to reduce the overheads of microservice-based communication?

Comprehensive studies by datacenter operators [46, 67] have shown that the datacenter tax is comprised of a set of procedures that are commonly used across many applications, which includes RPCs, serialization/deserialization, compression/decompression, cryptography (encryption, decryption, hashing) and memory management (memory allocations/deallocations, copies and moves). For instance, compression and memory management operations account for 6% and 10% of all Google's fleet-wide CPU cycles, respectively [46]. Similar observations were made at Facebook's datacenters [67], where caching services spend as much as 6% of their execution time performing data encryption. Fortunately, the interfaces exposed by these operations have, over time, evolved into highly stable and mature components. As a result, their functions are prime candidates for hardware acceleration.

Another consequence of decomposing applications into distributed microservices is that processing a single client request entails serially invoking multiple microservices, each of which must meet stringent latency requirements, or, risk adversely impacting user experience [33]. The research literature is rich with proposals to curb microservice tail latency, including work that (a) focuses on optimizing operating system threading designs [68], (b) uses machine learning techniques to efficiently manage resources while preserving SLOs [63, 73], (c) proposes the use of optimized serverless runtimes for microsecond-scale tasks [42] and (d) builds new hardware to mitigate the effects of queuing delays [58] and to exploit SIMT execution patterns commonly found in microservices [49]. Therefore, this thesis does not tackle the problem of minimizing tail latency. Instead, we explore how hardware accelerators can be best used to improve application throughput.

1.2 Accelerators

Having understood the potential of accelerators in mitigating performance overheads, we now look at accelerators which (1) are available in commodity hardware and (2) have been recently proposed in the research literature.

Accelerators can be broadly classified as follows:

1. ISA Extensions - As has often been the case, commonly used functions are implemented in hardware and made available to programmers as extensions to the processor's Instruction Set Architecture (ISA). While ISA extensions are not typically classified as "accelerators", they do provide domain-specific hardware acceleration. For instance, Intel's AVX [11], AMX [10] and AES-NI [9] augment the x86 ISA with instructions to perform vector operations, matrix multiplication and symmetric encryption respectively. More recently, Karandikar et al. [48] use per-core accelerators to extend the RISC-V ISA with instructions that speed up protobuf [25] serialization and deserialization.

2. On-chip accelerators - These accelerators reside on the CPU die and are typically shared between physical cores. Integrated GPUs [3], data streaming accelerators [12], load balancers [14], in-memory analytics accelerators [15] and accelerators for operations such as compression, encryption and memory copies found on modern chipsets [18, 19], are examples of on-chip accelerators. Accelerators found on modern SoC SmartNICs [21] are also on-chip accelerators that reside on the same die as the NIC's general purpose cores. Recent research has also explored the possibility of using on-chip NIC-integrated accelerators [41, 61, 51] to mitigate the performance overheads of Remote Procedure Calls (RPCs).

3. Off-chip accelerators - These are typically accessed using PCIe and are shared among NUMA nodes. Examples include discrete GPUs [20], TPUs [43], and SmartNICs [21].

1.3 Offload Latency

Offloading computation to hardware accelerators incurs a fundamental latency, which depends on a number of factors. Throughout this thesis, we refer to an operation's "offload latency" as the wall-clock time taken by an accelerator to execute the operation, including time needed to set up the operation. Offload latency determines whether applications that use accelerators achieve lower latencies i.e. if offload latency outweighs the latency of software computation, applications see no benefits. As a result, understanding the factors [28] that influence offload latency is critical to unlocking accelerator performance:

1. Size of Offloaded Data - Offload latency depends on the size of data offloaded i.e. its granularity. For small granularities, the overhead of offloading computation could potentially outweigh the benefits of acceleration, yielding no latency improvements. However, for larger granularities, the overhead is amortized, yielding overall speedups.

2. Accelerator Location - As discussed in Section 1.3, accelerators either reside *on-chip* or *off-chip*. Offloading computation to an on-chip accelerator incurs the lowest overheads because the accelerator resides on the same CPU die, eliminating overheads that result from traversing an off-chip interconnect. However, since on-chip accelerators are baked into CPU hardware, their capacity/count cannot be scaled up. In contrast, off-chip accelerators are typically accessed via PCIe and have microsecond-scale offload latencies. The number of off-chip accelerators can be scaled up to match the number of available PCIe slots, making it easier to alleviate performance bottlenecks arising from accelerator overload.

3. Accelerator API - The accelerator's API determines how application software interacts with it. For example, developers may choose a framework that provides a higher level of abstraction such as DOCA [23], or a lower level interface like DPDK [13]. While higher-level APIs may ease software development, we expect them to trade performance for ease-of-use.

4. Computation Complexity - Kernels offloaded to accelerators can have different complexities ranging from sub-linear to super-linear. We expect complex algorithms to have a higher execution cost than simpler ones.

In this thesis, we study the effect of (1) offload size and (2) accelerator location on offload latency; we defer investigating the accelerator's API and kernel complexity to future work. In the next section, we implement benchmarks to measure the offload latency of a compression accelerator available on Nvidia Bluefield SmartNICs [21].

Chapter 2

Performance of a Compression Accelerator

Despite the plethora of accelerators available today, it's unclear how they can be best used to improve application performance. Towards this end, accelerometer [67] develops an analytical framework to estimate accelerator speedup by using a variety of parameters, including setup time, offload size and queuing delays, whose values are assigned based on accelerator hardware specifications. However, the authors note that certain parameter assignments may not be completely accurate. For instance, their experiments assume that offloading all encryption operations to an accelerator yields speedups, while observing the possibility that only certain granularities benefit from hardware offload. Is it really beneficial to offload all computation to hardware, or, do only requests with certain characteristics benefit from hardware offload? We attempt to answer this question by characterizing the performance of a compression accelerator found on Nvidia Bluefield SmartNICs.

Experimental Setup

We perform our experiments on CloudLab's r7525 instances, equipped with Nvidia Bluefield 2 SmartNICs that support hardware accelerators for a variety of operations including compression, encryption and hashing. We develop a simple benchmark which varies the size of data offloaded and measures the corresponding offload latencies for Bluefield's compression accelerator, which supports the Deflate compression algorithm [5]. The latency of compression

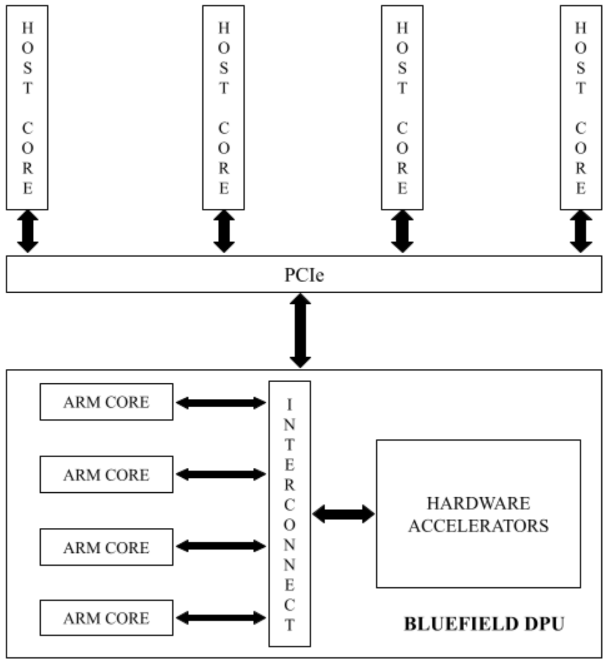

Figure 2.1. Experimental Setup

Input Type	Description
HTML	HTML source code for a webpage on the internet
Numeric	NYC Yellow Taxi Trip Data [24]
Random Numbers	Randomly generated integers between 0 and 255
English Text	Multiple paragraphs of real English text
Repeated Character	A string consisting of a single character repeated multiple times

Table 2.1. Input Types

algorithms varies widely depending on the type of data being compressed. To better understand the relationship between input data and compression latency, we study a variety of possible input data, as described in Table 2.1. Two of these input datasets, namely "Random Numbers" and "Repeated Character" are not realistic; however, we include them to provide upper and lower bounds, respectively, on compression latency.

We perform experiments to measure the offload latency for both *on-chip* (offload from Bluefield's ARM cores) and *off-chip* (offload from host's x86 cores) accelerator configurations, which are measured by configuring the Bluefield to use "ECPF" (Embedded CPU Function) and

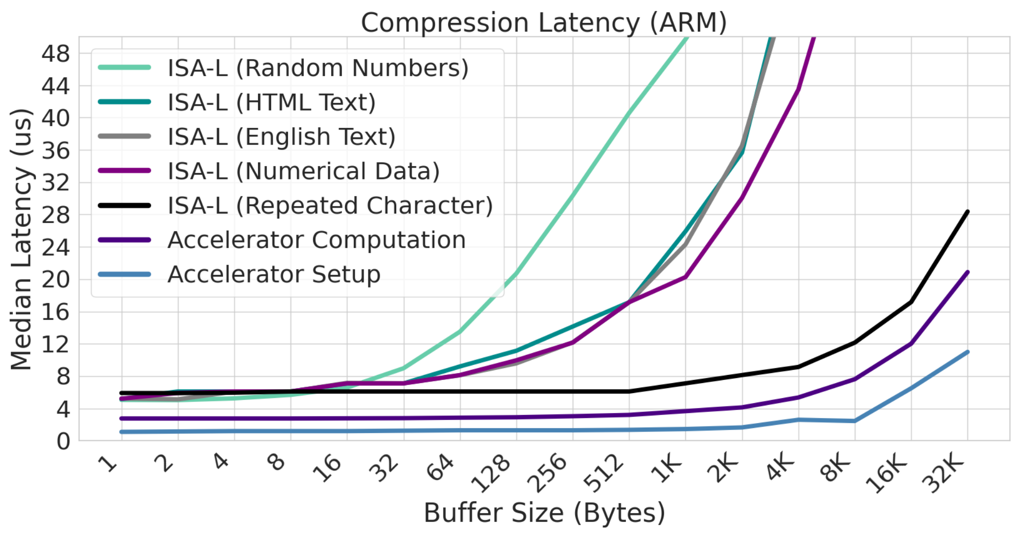

Figure 2.2. Median Compression Latency from NIC Cores

"Separated-Host" modes [22] respectively. Figure 2.1 illustrates our setup.

We present our findings using two graphs each for both on-chip and off-chip offload. The first of these graphs contains a detailed breakdown of software compression latencies for each kind of input data evaluated, whereas the second graph shows the median offload latency, flanked by shaded regions whose extremities correspond to the minimum and maximum values across different inputs. Our observations are as follows:

1. Compression latency with the ISA-L software library varies significantly depending on the input data, as shown in Fig. 2.2 and Fig. 2.4. However, this is not the case for the hardware accelerator - compression latency remains relatively unaffected by the input data.

2. On-Chip Offload - The results in Fig. 2.2 and Fig. 2.3 show that software compression latency always exceeds the accelerator computation latency, irrespective of the nature of input data; this is because the Bluefield's wimpy ARM cores aren't well suited to perform computationally intensive tasks such as compression. Therefore, offloading compression operations to an on-chip integrated accelerator yields lower latencies than software compression for all inputs and request sizes.

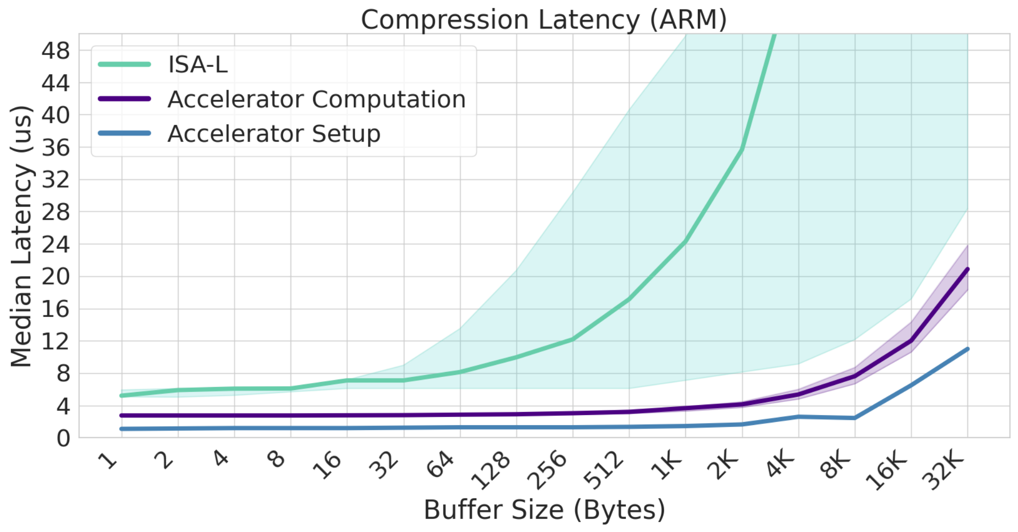

Figure 2.3. Median Compression Latency from NIC Cores

3. Off-Chip Offload - The results in Fig. 2.4 and Fig. 2.5 show that the median software compression latency for small buffers ($\leq$ 256 bytes) is lower than the median accelerator computation latency, as indicated by the gray region in Fig. 2.5. Therefore, compressing small buffers in software using beefy x86 cores typically yields lower latencies as compared to hardware offload. However, for large buffers ($>$ 256 bytes), hardware offload yields lower latencies, as indicated by the graph's white region.

4. The accelerator's setup time for on-chip offload lies in the range 1 μs - 11 μs and for off-chip offload, between 360 ns and 2.8 μs. For both on-chip and off-chip offload, the setup time remains lower than all other measured values. This implies that applications whose requests aren't latency-critical can optimize for throughput by always offloading computation to the accelerator (discussed further in Chapter 3.2).

Discussion

Comparing the median "Accelerator Computation" latencies (purple lines) in Figures 2.3 and 2.5, we see that for all input buffer sizes, the difference between on-chip and off-chip offload latencies is only a couple of microseconds. This difference can be attributed to PCIe's round-

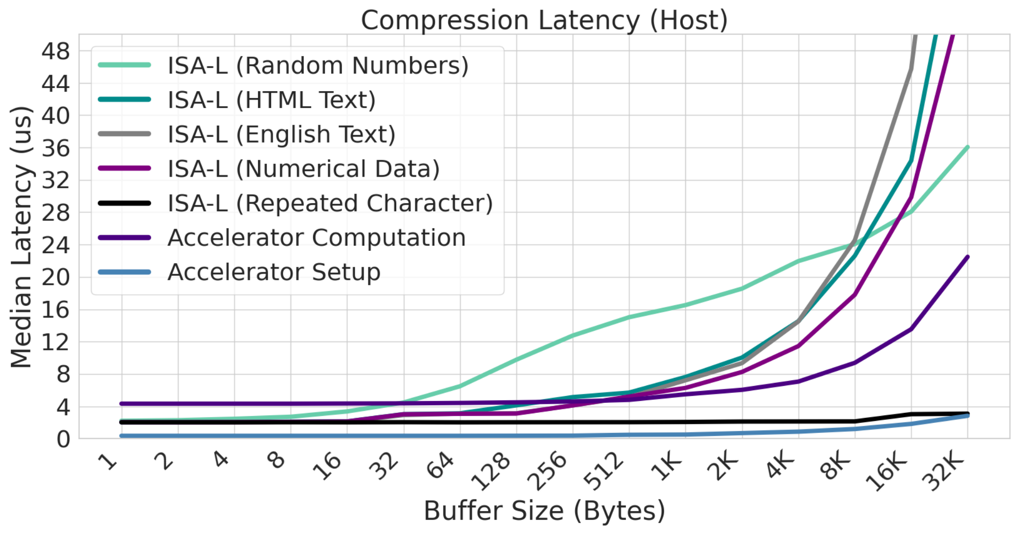

Figure 2.4. Median Compression Latency from Host Cores

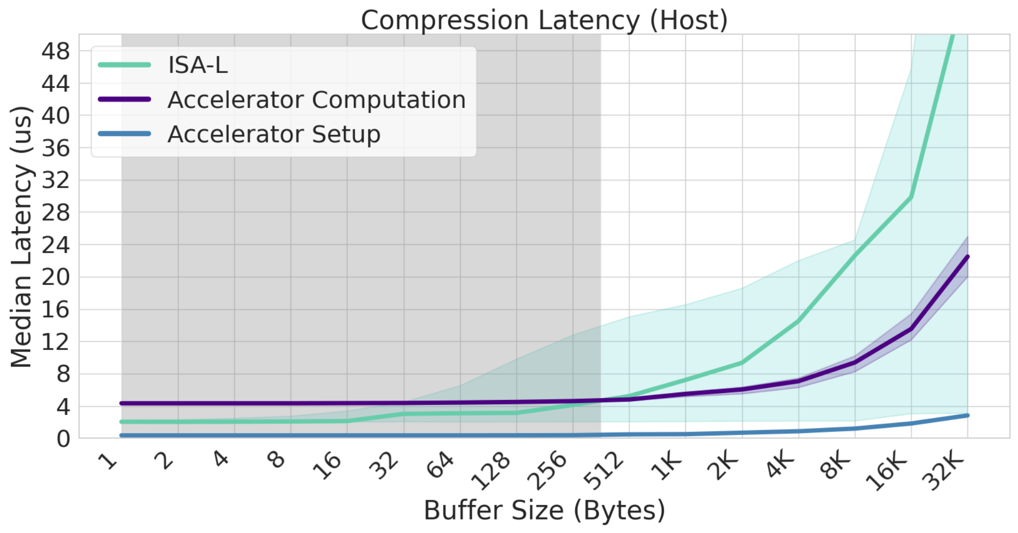

Figure 2.5. Median Compression Latency from Host Cores

trip time, which contributes to a significant fraction of the overall offload latency for smaller buffers. While applications that are completely offloaded to a SmartNIC's general purpose ARM cores [50, 54, 65] could benefit from access to on-chip accelerators, we expect this approach to be infeasible for applications with massive memory footprints i.e. the order of a hundred gigabytes [31]. As a result, we only consider off-chip acceleration in the rest of this thesis. For buffers larger than 8 KB, PCIe's costs are amortized by high offload latencies, indicating the viability of off-chip compression acceleration. Further, we find that small buffers achieve low compression latencies with highly optimized software libraries, whereas large buffers require hardware acceleration. Could we exploit these insights for better performance? We address this question in the next section.

Chapter 3

Design and Implementation

In this section, we discuss the design and implementation of our proposed system, which must make two key scheduling decisions: (a) when computation must be offloaded to hardware, and (b) whether offload must be performed asynchronously or synchronously. We first present the design goals we would like to achieve, followed by an overview of the system's design and conclude with details about our implementation.

3.1 Design Goals

1. Minimize Runtime Overheads - Offloading computation to hardware must be done as efficiently as possible, eliminating all intermediate overheads that could adversely affect offload latency.

2. Transparent Execution - Application software must be able to issue acceleration requests using standard programming interfaces. Moreover, our system must be able to automatically decide whether an operation should be offloaded or not, using information about the accelerator's current load as well as the application's characteristics e.g. whether it is latency-critical.

In subsequent sections, we present the aspects of our system's design and implementation that help us achieve the aforementioned goals.

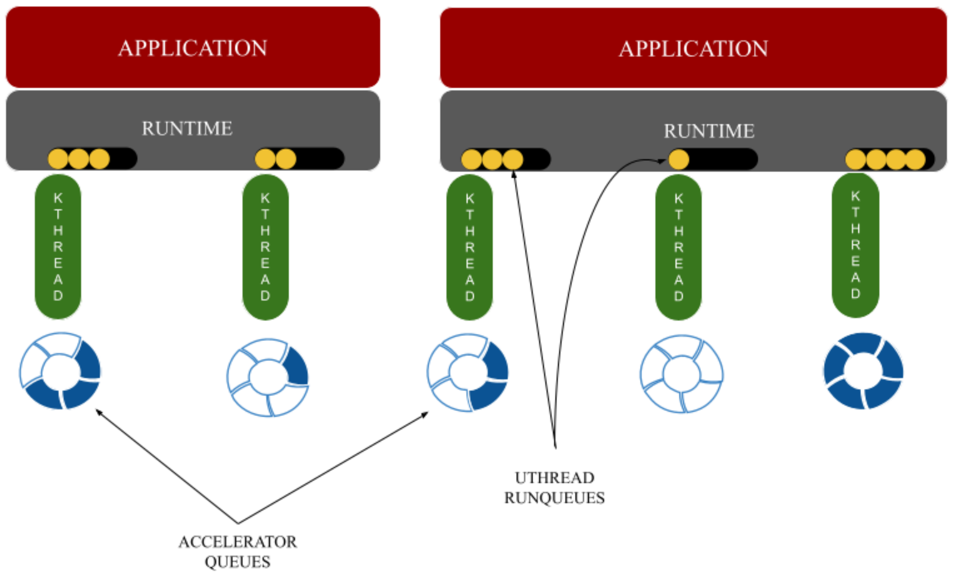

Figure 3.1. System Design for Hardware Accelerator Offload

3.2 Design Overview

Figure 3.1 presents the key components of our system, which shares architectural similarities with Caladan [37] and is designed to run in a standard Linux environment. Each application managed by our system runs as a normal Linux process, linked with a runtime that provides useful abstractions such as threads, mutexes, condition variables, network sockets and acceleration libraries. When an application begins execution, it initializes its runtime with a set of parameters that indicate its performance requirements, such as required number of CPU cores, latency-criticality, the need for garbage collection, etc. The runtime spawns a new kernel thread for each CPU core the application is allocated, and balances load across kernel threads using work stealing, which has shown to yield excellent results for micro-second workloads [57, 62].

Application logic runs in lightweight user-level threads, which are scheduled on kernel threads (each of which has its own local runqueue) by the runtime scheduler. Switching to a different thread of execution while awaiting completion of a high-latency operation (such as accelerator I/O) yields throughput benefits if the overhead of thread context switches is much smaller than the operation's execution latency. From Chapter 2, we know that the Bluefield's compression offload latency is on the order of a few microseconds. To improve the throughput of a multithreaded application that uses the Bluefield's compression accelerator, our system must support a thread context switch latency that is at least an order of magnitude faster i.e. nanosecond-scale context switching latency. Our choice of user-level threads was motivated by the need for fast context switches, which Caladan's user-thread library achieves in approximately 50 ns (one-way). Stackless coroutines, a viable alternative to user-level threads, could further reduce context switching overheads to just 12 clock cycles [72] - we leave exploring this option to future work. In line with design goal $\textcircled{1}$, runtimes have direct access to accelerators through kernel bypass libraries, thereby eliminating the overheads of traversing the Linux kernel. Handling accelerator I/O in user-space also gives runtimes more insight into accelerator resource usage, enabling the implementation of intelligent scheduling policies.

Runtime Scheduler

The runtime scheduler is responsible for (1) multiplexing an application's user-threads onto its kernel threads, (2) deciding whether an operation must be offloaded to hardware or not, and (3) deciding whether hardware offload must be performed synchronously or asynchronously. To meet these goals, the scheduler must be able to (a) quickly switch between user-threads and (b) use information about an application's performance requirements, as well as an accelerator's current load while making scheduling decisions. Our scheduler is designed to use the insights obtained in Chapter 2 to offload computation to hardware accelerators as efficiently as possible. We leave the implementation of policies/mechanisms that make scheduling decisions based on accelerator load to future work.

<u>Mechanism</u> - Our runtime exposes a simple synchronous API to applications, and dynamically chooses the best scheduling policy under the hood. In keeping with design goal $\textcircled{2}$, application developers don't have to work with convoluted asynchronous programming

models, or worry about the minutiae of scheduling optimizations. For asynchronously offloaded requests, the scheduler submits an operation to the accelerator and parks the issuing user-thread so that other user-threads may continue execution in the interim time period. Upon completing a request, the accelerator enqueues its results onto a software queue; the scheduler periodically polls the software queue for completion updates and wakes up user-threads as their requests get completed.

Policy - We consider two classes of applications - (a) latency-critical (LC) and (b) batch. Our policy caters to the application's performance requirements, and chooses how to best service a compression request based on the following heuristics (summarized in Table 3.1):

1. Latency-Critical Apps - Requests must be processed as quickly as possible. From Chapter 2, we know that for small buffers, the accelerator's compression latency exceeds software compression latency; hence, we achieve low latency by always compressing small buffers in software. In contrast, the software compression latency for large buffers outweighs the accelerator's compression latency, due to which, our system offloads all large compression requests to the accelerator. However, should operations be offloaded synchronously or asynchronously? We choose synchronous offload (where the issuing user-level thread spin polls the accelerator for completion) to minimize latency.

2. Batch Applications - Requests aren't latency-critical and we therefore optimize for throughput. For all buffer sizes considered, the following statements hold true: (a) the accelerator's setup latency is smaller than the software compression latency, and (b) the accelerator's computation latency is much larger than its setup latency. As a result, we maximize throughput by asynchronously offloading all compression requests to the accelerator.

The desire to support both synchronous and asynchronous offload within the same application gives rise to an interesting problem. Consider a scenario where user-threads u_1 and u_2 issue compression requests r_1 and r_2 respectively. r_1 is a large request that isn't latency-critical

	Small Buffers	Large Buffers
Latency-Critical	Software	Sync Offload
Batch	Async Offload	Async Offload

Table 3.1. Runtime Scheduling Policy for Compression Requests

and is therefore asynchronously offloaded. However, r_2 is a large latency-critical request that must be synchronously offloaded. Suppose r_1 is submitted to the accelerator first. Since r_1 is offloaded asynchronously, the scheduler will issue r_1 to the accelerator and subsequently park u_1 so that u_2 may execute. Now, since r_2 is latency-critical, u_2 issues r_2 to the accelerator and spin-polls its software queue, awaiting completion. However, it is possible that r_1 completes before r_2, causing u_2 to incorrectly dequeue r_1's completed operation from the software queue. To prevent this problem, our system does not allow individual application threads to be configured with different performance requirements. Instead, the entire application must either be "Batch" or "Latency-Critical", such that all requests requiring hardware acceleration are either offloaded synchronously (for latency-critical) or asynchronously (for batch).

3.3 Implementation

Our implementation is built on the open-source release of Caladan [37], which was a good starting point due to its support for user-level threads and kernel-bypass I/O. Similar to Caladan, we augment our runtime with DPDK's poll mode drivers for fast, direct access to accelerators from user-space (design goal ① above). It is worth noting that integrating DPDK with our runtime allows client applications direct access to DPDK APIs, which could be used to circumvent the runtime's APIs. However, such use will only lead to performance degradation, and is therefore strongly discouraged.

Our runtime implements a compression library that exposes a familiar, synchronous API to developers. The library intelligently decides whether to offload computation to hardware, or, to perform it in software, based on each request's unique characteristics. This library is implemented in around 1500 lines of C code (including comments) and currently includes support for (1) Nvidia Bluefield hardware acceleration and (2) software compression with ISA-L [16].

For efficient asynchronous offload, the runtime scheduler must be able to quickly check whether any pending operations have completed. Unfortunately, DPDK's API currently does not support checking for completions without removing completed operations from the accelerator's software queue i.e. to retrieve the number of completed operations, an application must use the function `rte_compressdev_dequeue_burst()`, which returns the number of completed operations, but also removes them from the device's queue. To circumvent this problem, we add around 30 lines of C code to DPDK's MLX5 driver, which checks a device queue's completion status in O(1) time, without dequeuing completed operations. Our patch simply inspects the queue's producer and consumer indices and returns the arithmetic difference between them i.e. if the producer's index is ahead of the consumer's index, operations are available for processing.

The limitations of our current implementation are as follows. First, we only support offloading computation from host cores over PCIe because Caladan cannot run on ARM processors - context switches between user-threads rely on architecture-specific assembly code, which is currently implemented only for the x86 ISA. We leave the implementation of ARM context switches i.e. support for on-chip offload from Bluefield ARM cores, to future work. Second, though applications may use multiple user-threads, runtimes are limited to a single processor core. The use of multiple cores per runtime can be achieved by allocating separate accelerator queues to each runtime - we are currently working towards implementing this feature.

Chapter 4

Evaluation

In evaluating our system, we wish to answer the following questions:

1. How do the different compression modes and compression policies compare against each other with respect to latency per operation and overall application throughput?

2. How do our system parameters, namely polling interval dequeue batch size, affect the performance of asynchronous offload?

4.1 Experimental Setup

We perform our experiments on CloudLab's r7525 instances. Each server is a dual-socket NUMA machine with two 32-core AMD 7542 processors (with Simultaneous Multithreading) clocked at 2.9GHz, 512GB ECC DDR4 DRAM clocked at 3200MHz, a Dual-port Mellanox ConnectX-5 25 Gbps NIC and a Dual-port Mellanox BlueField2 100 Gbps SmartNIC. The Bluefield contains 16GB DDR4 DRAM and 8 ARM A72 cores which share access to on-chip hardware accelerators for a variety of operations including compression, encryption and hashing. Our experiments use closed loop load generators, wherein a new request is not generated until a pending request has completed execution. Moreover, all experiments are run with application runtimes that have been allocated a single CPU core.

We model the behaviour of datacenter applications that frequently compress data by developing a benchmark to generate compression buffers whose sizes follow the distribution of

Compression Mode/Policy	Description
Software	All compression requests are handled synchronously in software using the ISA-L [16] compression library
Sync	All compression requests are synchronously offloaded to the Bluefield's compression accelerator. Each request's user-thread busy polls the accelerator for completion, minimizing delay
Custom (LC)	For latency critical applications, buffers smaller than (or equal to) 256 bytes are compressed synchronously in software using the ISA-L [16] compression library. However, buffers larger than 256 bytes are synchronously offloaded to the Bluefield's compression accelerator
Async	All compression requests are asynchronously offloaded to the Bluefield's compression accelerator
Custom (Batch)	For batch applications, all compression requests are offloaded asynchronously to the Bluefield's compression accelerator because the accelerator's setup latency is much smaller than the software compression latency

Table 4.1. Compression Modes and Policies

request sizes in Facebook's production workloads [67]. Our benchmark spawns 100 user-threads, each of which submits 100 compression requests (for a total of 10,000 requests) modeled after the specific application's buffer size distribution. We study the distributions of two such applications, "Cache-1", a distributed-memory object caching service, and "Feed-1", a microservice in Facebook's News Feed service that calculates the relevance of news stories for each user. The CDFs of compression buffer sizes for these two applications are shown in Fig. 4.1.

4.2 Compression Modes

We evaluate five different compression modes, whose details are summarized in Table 4.1. The remainder of this section compares the various compression modes studied thus far and evaluates their effect on application throughput and latency.

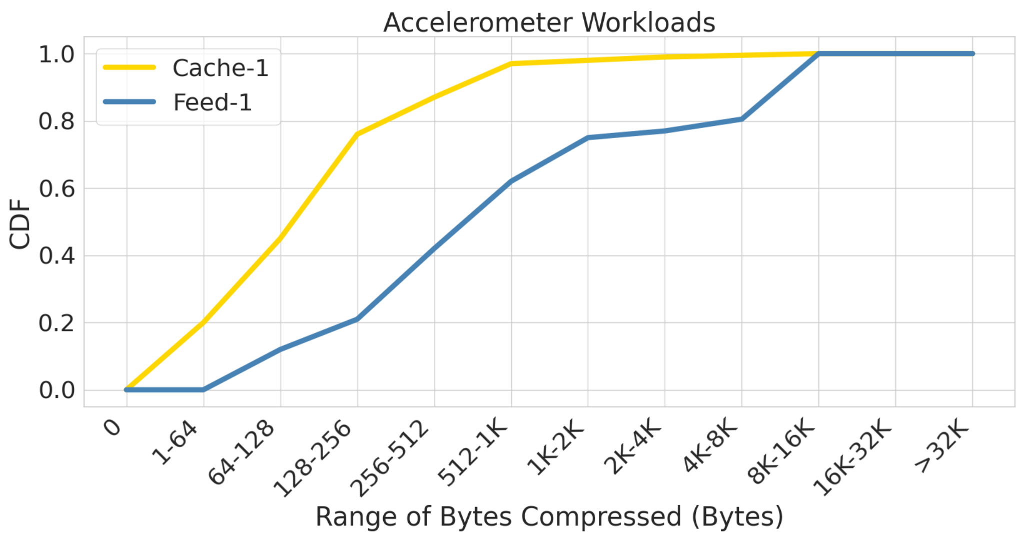

Figure 4.1. Accelerometer Request Size Distributions

4.2.1 Datacenter Workloads

In this section, we evaluate our system's performance on the Cache-1 and Feed-1 workloads described above. To account for the various types of data that input buffers could possibly contain, we run experiments with both HTML and numeric data. In evaluating performance, we consider two metrics: (a) per-request latencies and (b) overall application throughput. Since both types of input data produced similar results, we only include graphs for HTML data.

We first examine the distribution of per-request latencies for Cache-1, shown in Fig. 4.2. For every compression mode, more than 85% of all requests complete under 10 μs. In particular, "Sync" and "Custom (LC)" achieve the best latency distributions because the user thread issuing the request busy polls the accelerator for completion, whereas "Async" and "Custom (BE)" achieve the worst distributions because user-threads issuing requests are suspended until the scheduler is subsequently invoked. Though "Custom (LC)" compresses small buffers in software to further minimize latency, the effect of this optimization isn't pronounced because the fraction of Cache-1's requests that use small-sized buffers is low.

Next, we look at the latency distribution for Feed-1, shown in Fig. 4.3. We observe that

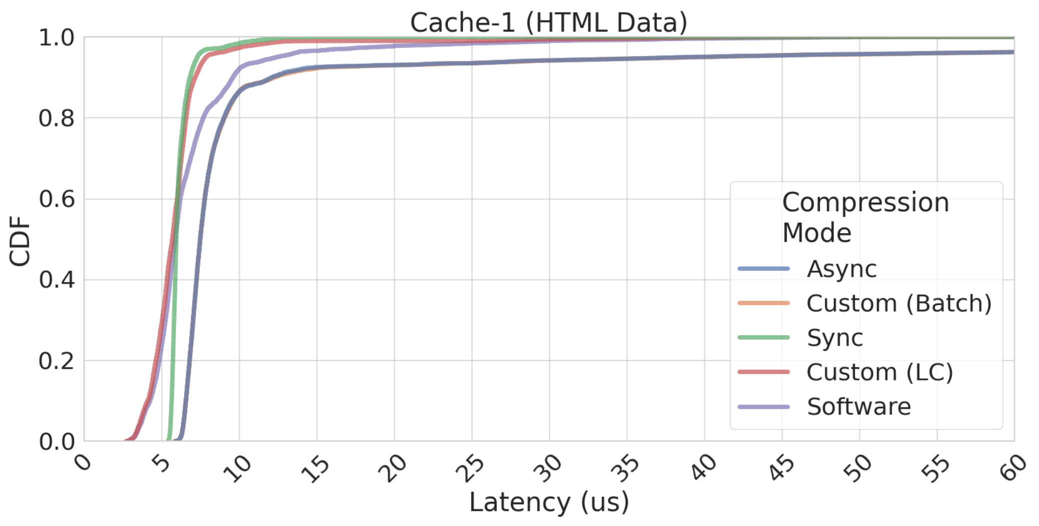

Figure 4.2. Cache-1 Latency CDF (HTML Data)

Feed-1 achieves a worse distribution than Cache-1 i.e. only 70% of its requests complete under 10 μs because Feed-1 uses larger buffer sizes, whose requests take longer to execute. However, similar to Cache-1, "Sync" and "Custom (LC)" achieve the best latency distributions, whereas "Async" and "Custom (BE)" achieve the worst distributions. Once again, the optimization employed by "Custom (LC)" is not visible because Feed-1's request size distribution doesn't contain small buffers.

Finally, the results for throughput are shown in Fig. 4.4. For both workloads, compression in software has the lowest throughput (143.7K req/s for Cache-1 and 68.6K req/s for Feed-1). Synchronous compression with "Sync" and "Custom (LC)" improves upon this, achieving 1.12-1.6x higher throughput because both workloads contain a significant fraction of large buffers, which benefit from hardware offload. Finally, asynchronous compression with "Async" and "Custom (BE)" achieves the highest throughput, 3.4-3.8x higher than software compression, because other user threads are allowed to make progress while some of them wait for completion. Interestingly, with asynchronous offload, Cache-1 achieves a 2x higher throughput as compared to Feed-1 because Feed-1 uses larger buffer sizes, whose requests take longer to complete.

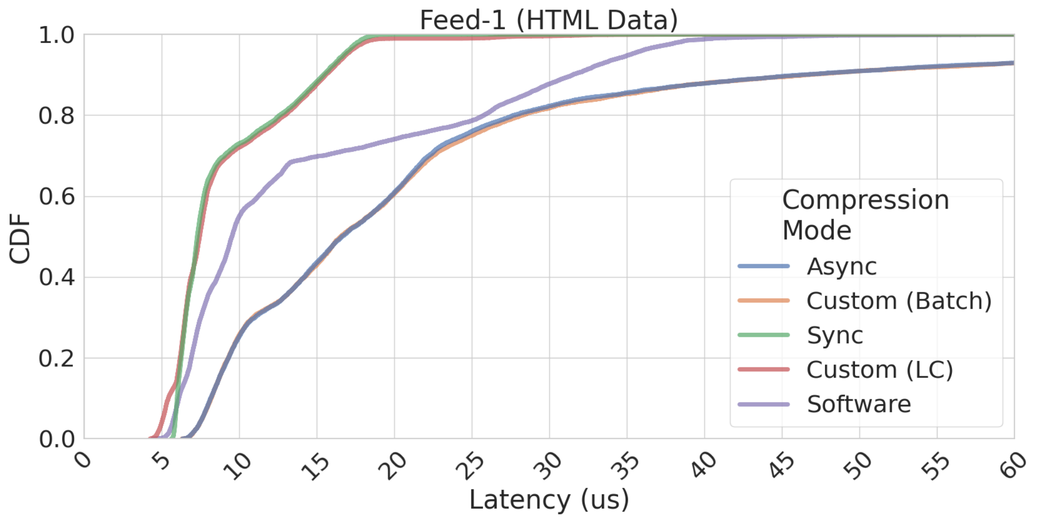

Figure 4.3. Feed-1 Latency CDF (HTML Data)

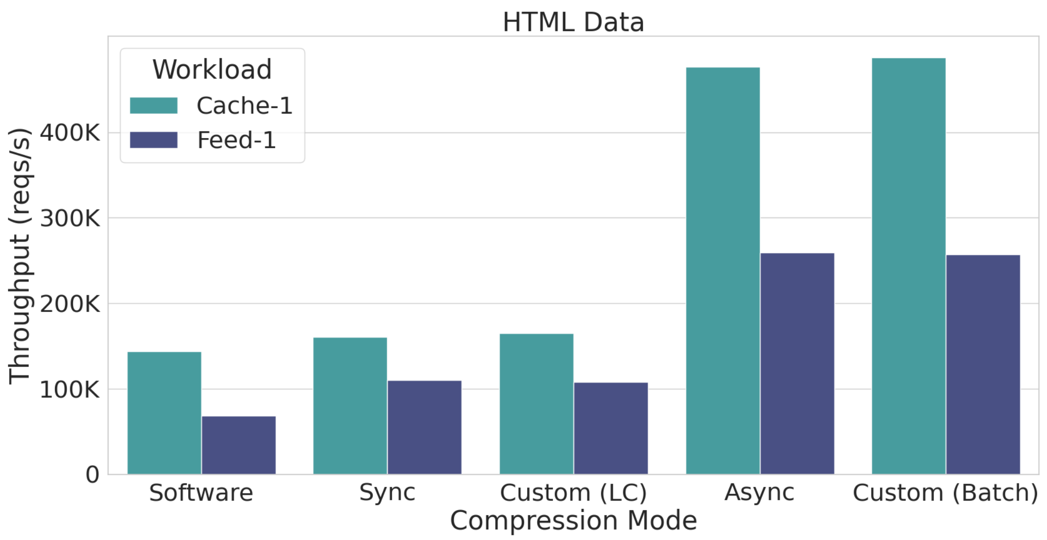

Figure 4.4. Throughput (HTML Data)

Compression Mode	Throughput (req/s)	Median Latency (μs)	90th Perc. Latency (μs)
Software (ISA-L)	28.1 K	38.4	42.4
Synchronous Offload	39.3 K	27.6	30.3
Custom (LC)	39.3 K	27.6	30.2
Asynchronous Offload	60.2 K	36.1	39.4
Custom (BE)	60.3 K	36.0	39.9

Table 4.2. Performance of a Synthetic Web Server

4.2.2 Synthetic Web Server

To understand how our system performs on real applications, we evaluate the per-request latencies (median and 90th percentile) and overall throughput of a synthetic web service, whose implementation was borrowed from the open-source release of AIFM [64], that encrypts and compresses 8KB objects before returning them to clients. The web service spawns multiple threads, each of which submits compression requests. Once a thread submits a request, it waits for the request to complete before submitting a new request. Therefore, it uses a closed loop load generator as well. The results are shown in Table 4.2. Compression in software yields the highest median latency and lowest throughput. Synchronous compression with "Sync" and "Custom (LC)" achieves the lowest latency, which is 28% lower than software compression, and a throughput improvement of 1.4x. Finally, asynchronous compression with "Async" and "Custom (BE)" achieves the highest throughput, which is 2.14x higher than software compression. Our custom policies that adaptively choose between software compression and hardware offload perform similar to policies that always offload compression to the accelerator because the web service only compresses large objects (8KB).

4.3 Polling Interval

Asynchronous offload relies on the runtime scheduler being able to periodically poll accelerator queues for completion events. The frequency at which the scheduler checks for completion events has a significant influence on per-request latency as well as overall application

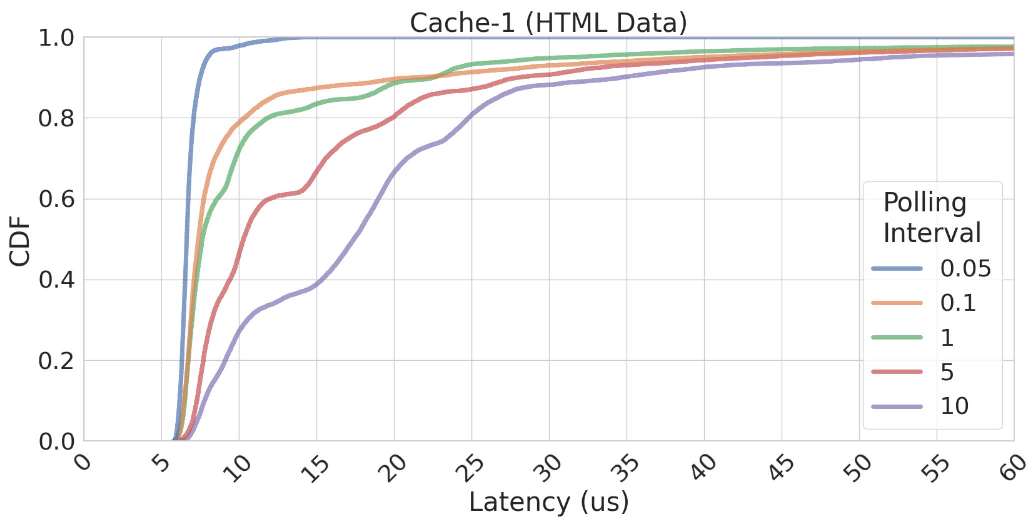

Figure 4.5. Polling Interval Latency CDF (HTML Data)

throughput. With more frequent polling, we ensure that requests aren't kept waiting for extended periods of time - therefore, we expect better latency distributions, at the expense of reduced throughput. With a lower polling frequency, requests could potentially be suspended for longer periods of time, resulting in high per-request latencies. However, polling less frequently allows application threads to get more work done, resulting in better application throughput. In this section, we quantify the effect of the scheduler's polling interval on per-request latencies, as well as application throughput. We configure the scheduler's polling interval statically. However, the actual polling interval achieved in practice may deviate from the statically assigned value. As a result, our graphs illustrate results for the "target polling interval", which represents the statically assigned value. Moreover, we only run experiments for Cache-1 because its buffer size distribution includes small buffers and is therefore more interesting than Feed-1. Finally, we note that the number of operations dequeued from the accelerator's queue during each polling iteration affects latency and throughput. As a result, we fix the dequeue batch size to 16 (based on experiments in the next subsection) so that we don't process more than 16 requests per polling iteration. Fig. 4.5 shows how polling interval affects per-request latency and Fig. 4.6 shows how it affects overall application throughput.

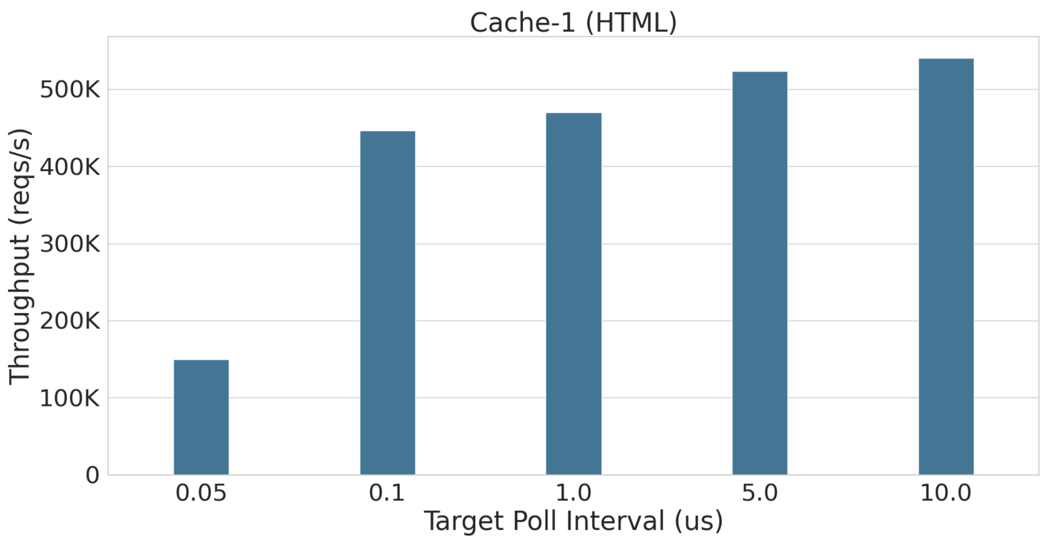

Figure 4.6. Polling Interval Throughput (HTML Data)

The graphs indicate that using an extremely low polling interval (0.05 μs) achieves the best latency distribution, however, suffers from severe throughput degradation. At the other end of the spectrum, the highest throughput is achieved with a large polling interval (10 μs), at the expense of a poor latency distribution. As a result, latency-critical applications should use a shorter polling interval, while expecting lower throughput. A reasonable middle-ground is a polling interval of 1 μs, which achieves a good balance between per-request latency and application throughput.

4.4 Dequeue Batch Size

As alluded to in the previous subsection, another factor that determines the latency and throughput of asynchronous offload is dequeue batch size i.e. the maximum number of operations processed each time the scheduler polls the accelerator's queue. With large dequeue batch sizes, we expect per-request latencies to suffer because the scheduler could, in theory, be busy processing completion events for a large number of operations before returning control to application threads. To investigate the effect of dequeue batch size on latency and throughput, we fix the scheduler's polling interval to 1 μs, which achieves a good balance between latency

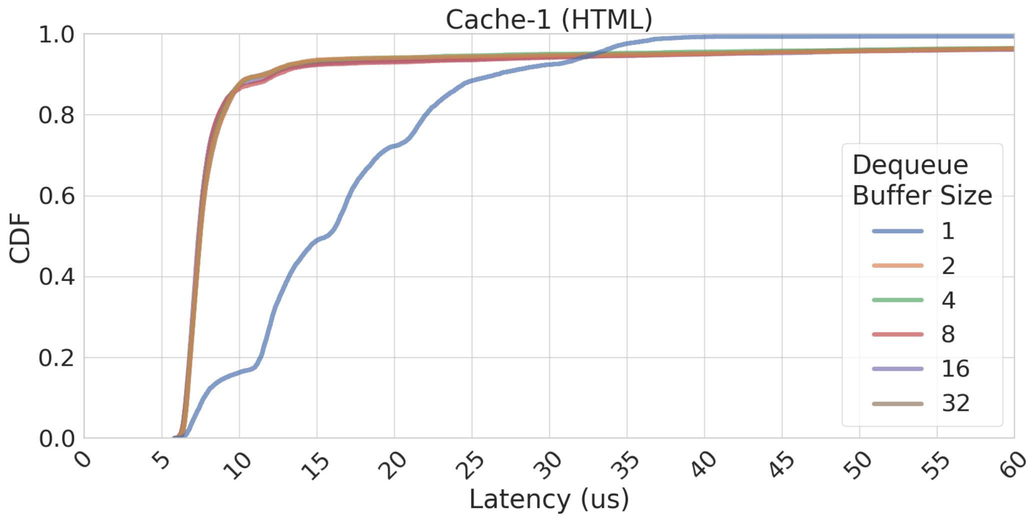

Figure 4.7. Dequeue Batch Size Latency CDF (HTML Data)

and throughput.

Per-request latency distributions are shown in Fig. 4.7. From the figure, it is clear that the dequeue batch size must be at least 2 operations to achieve low latency; with small dequeue batch sizes, the number of threads awoken per polling iteration of the runtime scheduler is very small, as a result of which, the system is more efficient when it can amortize the overheads of polling by polling multiple completions at once. Secondly, all dequeue batch sizes greater than or equal to two achieve similar latency distributions because on average, the number of completed operations available for processing during each poll interval is between two and four. As a result, larger batch sizes do not influence per-request latency. Another consequence of this phenomenon is that dequeue batch size does not affect application throughput.

Chapter 5

Related Work

5.1 Accelerators

Comprehensive studies [46, 67] have shown that datacenter applications rely on a set of common procedures that are highly amenable to hardware acceleration. While accelerators for operations such as compression and encryption are available in commodity hardware today [21], recent work proposes custom hardware to alleviate other components of the datacenter tax. Cerebros [61] proposes offloading the entire RPC layer to an on-chip, shared NIC-integrated accelerator that uses affinity-based request steering to improve performance. nanoPU [41] takes the idea of NIC-CPU co-design to its extreme by delivering RPC requests directly to CPU registers. In contrast to shared accelerators, Karandikar et al. [48] propose the design of a private per-core accelerator for serialization/deserialization operations (performed by protocol buffers [25]) to better support applications that don't use RPCs, e.g., storage. In this thesis, we take a step towards understanding the performance characteristics of such emerging accelerators and explore the role that operating system scheduling plays in optimizing their usage.

GPUs have traditionally been used to accelerate highly parallel applications such as computer graphics and scientific workloads. More recently, GPUs and custom ASICs [43] have become increasingly important in providing the raw computational power needed to train deep neural networks [66]. Orthogonal work [38, 39] uses GPUs to preserve the flexibility/affordability of software routers by exploiting the inherent parallelism found in router applications for higher packet processing throughput. We do not consider GPUs in this thesis because commonly used GPU kernels for deep learning and cryptography have millisecond-scale offload latencies [53], which aren't well-suited for microsecond scale tasks.

Finally, introducing hardware accelerators in production is a non-trivial task, involving design, testing and planning to match expected load. Analytical modeling techniques play an important role in projecting accelerator speedup before expending resources for real hardware. LogCA [28] builds on prior research to develop a simple model based on a few parameters for accelerators. However, LogCA assumes that all hardware offload is synchronous. Accelerometer [67] overcomes this limitation to realistically model microservice speedup by capturing the concurrency that results from asynchronous offload. Similarly, we use asynchronous offload to improve the throughput of batch applications and evaluate our system using Accelerometer's request size distributions.

5.2 Dataplane Operating Systems

Faster networks and storage devices have shifted performance bottlenecks to systems software. Many systems circumvent this problem by separating the OS' dataplane from its control plane [32, 60], an idea that dates back to the Exokernel [34]. These systems typically achieve better performance by eliminating the overheads associated with traversing the OS kernel and processing interrupts. Snap [56] puts these ideas into practice, while retaining Linux as its control plane. In a similar vein, we use DPDK [13] to side step the Linux kernel and directly access accelerators from user-space. More recently, Demikernel [72] proposes using library OSes to abstract away the heterogeneity of dataplane systems by exposing a uniform, general-purpose application programming interface.

5.3 Operating System Scheduling

Scheduling mechanisms found in commodity OSes incur overheads on the order of milliseconds [52], and are hence unable to meet the microsecond-scale latency requirements of datacenter applications [30]. Recent research has explored the design of OS schedulers for requests that demand microsecond latencies. ZygOS [62] deviates from traditional data-plane OSes by using work stealing to avoid head-of-line blocking, paving the way for microsecond scale computing on multi-core servers. Shinjuku [44] uses fast preemption as a mechanism to develop efficient policies for workloads with highly dispersed request times. Demikernel [72] uses coroutines to achieve fast context switches. In contrast, we achieve low-latency context switches with the help of a cooperative user-level threading runtime based on Caladan [37], which multiplexes application user-threads on top of kernel threads. Context switches between OS kernel threads incurs a high latency that results from the need to switch between user and kernel execution modes; therefore, we rely exclusively on user-level threading for low latency. Ghost [40] eschews user-threads in favour of kernel threads because user-threading runtimes do not have control over when a kernel thread is scheduled, or, which CPU it runs on. Scheduler activations [29] circumvents this problem by notifying application runtimes about kernel scheduling events, allowing them to react optimally. However, our system differs from Scheduler Activations in that we do not allow user-threads to perform blocking I/O operations - instead, they are expected to use the runtime's APIs.

Commodity OS kernels use generic scheduling policies that cater to a wide range of workloads, resulting in sub-optimal performance for important applications [55]. OS kernel schedulers also make it difficult to quickly implement and test new scheduling policies across a large fleet of machines. As a result, recent research has proposed delegating scheduling policies to user-space processes. Syrup [45] allows the specification of scheduling policies for a wide range of system resources on a per-application basis. Ghost [40] uses user-space "agents" to instruct the kernel on how to schedule kernel threads on CPU cores.

In addition to high performance, CPU efficiency is of utmost importance in the datacenter [48, 70]. Shenango [59] uses thread and packet queueing delays as signals of impending SLO violations and uses fast core reallocations to avoid the overheads of dedicating CPU cores to application runtimes. Caladan [37] boosts CPU efficiency by quickly reacting to signals of interference between co-located tasks, thereby eliminating the need for static resource partitioning. McClure et al. [57] explore efficient load balancing and core reallocation policies that strike the best balance between CPU efficiency and tail latency.

5.4 SmartNIC Offload

The emergence of SmartNICs as additional sources of compute has exposed opportunities to improve host CPU utilization. SmartNICs can be built from a wide range of technologies, including ASICs, FGPAs and SoCs, each with its own tradeoffs. ASICs have functionality baked into hardware, offering the best performance at the cost of poor programmability. On the other hand, FPGAs improve programmability while providing performance close to ASICs. Finally, SoC SmartNICs offer the best programmability, albeit with poorer performance resulting from the use of general purpose cores. AccelNet [36] eschews multicore SoC SmartNICs due to their poor scalability and instead uses FPGAs for high-performance offload of end host network functions. In contrast, iPipe [54] shows that distributed applications with complex data structures cannot be efficiently offloaded to FPGA SmartNICs, and instead, use an actor-based framework to dynamically offload them to SoC SmartNICs. FlexTOE [65] eliminates host TCP data-path processing by offloading it to SmartNICs, while enabling on-the-fly customization at high speeds with fine-grained data parallelism. LineFS [50] uses similar ideas to offload the processing-intensive parts of a distributed file system. On the other hand, Lynx [69] offloads both the network control and data planes to SmartNICs, enabling direct communication between accelerators. Similar to the above research, we selectively offload parts of an application (that require acceleration) to multicore SoC SmartNICs. However, we only make use of the

SmartNIC’s accelerators, and do not use its general purpose programmable cores.

Chapter 6

Remaining Research Challenges

Multicore Scalability

In our current implementation, each runtime is configured to use a single CPU core. As a result, our benchmarks are not able to saturate the accelerator. We are working towards implementing support for multiple CPU cores per runtime, which would help us better understand accelerator scalability. If adding multicore support to runtimes results in accelerator saturation, the runtime scheduler's policies must be adapted to factor in accelerator load while making scheduling decisions.

Stackless Coroutines

Our runtime scheduler relies on the efficiency of user-threads to achieve a low context switching latency of 50ns. However, as noted in Chapter 3.2, context switches between stackless coroutines can be achieved in just 12 clock cycles [72]. We therefore plan to explore the use of stackless coroutines as an alternative to user-threads.

Factors Affecting Offload Latency

This thesis only studies how offload data size and accelerator location influence offload latency. We are investigating how other factors such as the accelerator's API and the offloaded kernel's computational complexity affect offload latency.

Accelerators

We exclusively study the performance of a compression accelerator found on Bluefield SmartNICs. We are working towards understanding the performance of a wider range of accelerators [12, 14, 15, 17, 18, 21].

Chapter 7

Conclusion

As hardware accelerators become more prominent in mitigating performance bottlenecks, deeply understanding their behaviour is crucial to unlocking their potential. In this thesis, we have evaluated the characteristics of a compression accelerator found on recent hardware and shown that for the workloads evaluated, always offloading compression to hardware yields the best results. More importantly, the choice between synchronous and asynchronous offload plays an key role in determining an application's throughput and per-request latency. Synchronous offload achieves the lowest latency, whereas, asynchronous offload achieves the highest throughput. Therefore, OS schedulers that understand the nature of workloads being accelerated, as well as characteristics of the accelerators being used, are well-poised to tackle the challenges faced by systems infrastructure in a post-Moore era.

Bibliography

[1] Adopting Microservices at Netflix: Lessons for Architectural Design. https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/.

[2] Amazon Prime Video Monitoring Service. https://tinyurl.com/4em5xd8.

[3] AMD Ryzen Radeon. https://www.amd.com/en/processors/ryzen-with-graphics.

[4] AWS Nitro. https://aws.amazon.com/ec2/nitro/.

[5] Deflate Compression. https://www.w3.org/Graphics/PNG/RFC-1951.

[6] Facebook Thrift. https://github.com/facebook/fbthrift.

[7] From a Monolith to Microservices + REST: the Evolution of LinkedIn's Service Architecture. https://www.infoq.com/presentations/linkedin-microservices-urn/.

[8] gRPC. https://grpc.io/.

[9] Intel Advanced Encryption Standard Instructions (AES-NI). https://www.intel.com/content/www/us/en/developer/articles/technical/advanced-encryption-standard-instructions-aes-ni.html.

[10] Intel Advanced Matrix Extensions. https://www.intel.com/content/www/us/en/products/docs/accelerator-engines/advanced-matrix-extensions/overview.html.

[11] Intel Advanced Vector Extensions (AVX). https://www.intel.com/content/www/us/en/architecture-and-technology/avx-512-overview.html.

[12] Intel Data Streaming Accelerator. https://01.org/blogs/2019/introducing-intel-data-streaming-accelerator.

[13] Intel DPDK. https://www.dpdk.org/.

[14] Intel Dynamic Load Balancer. https://www.intel.com/content/www/us/en/developer/articles/technical/proof-points-of-dynamic-load-balancer-dlb.html.

[15] Intel In-Memory Analytics Accelerator. https://www.intel.com/content/www/us/en/events/on365/intel-in-memory-analytics-accelerator.html.

[16] Intel Intelligent Storage Acceleration Library (ISA-L). https://github.com/intel/isa-l.

[17] Intel I/OAT. https://www.intel.com/content/www/us/en/wireless-network/accel-technology.html.

[18] Intel Quick Assist Technology (QAT). https://www.intel.com/content/www/us/en/architecture-and-technology/intel-quick-assist-technology-overview.html.

[19] Intel Sapphire Rapids. https://ark.intel.com/content/www/us/en/ark/products/codename/126212/products-formerly-sapphire-rapids.html.

[20] Nvidia A100 GPU. https://www.nvidia.com/en-us/data-center/a100/.

[21] Nvidia Bluefied 2 DPU. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-2-dpu.pdf.

[22] Nvidia Bluefied Modes of Operation. https://docs.nvidia.com/networking/display/BlueFieldSWv35011563/Modes+of+Operation.

[23] Nvidia DOCA. https://developer.nvidia.com/networking/doca.

[24] NYC Yellow Taxi Trip Data. https://data.cityofnewyork.us/Transportation/2018-Yellow-Taxi-Trip-Data/t29m-gskq.

[25] Protocol Buffers. https://protobuf.dev/.

[26] Sharing Modules Across Experience Services and Multi-Screen Applications. https://tech.ebayinc.com/engineering/sharing-modules-across-experience-services-and-multi-screen-applications/.

[27] What is Microservices Architecture? https://cloud.google.com/learn/what-is-microservices-architecture.

[28] Muhammad Shoaib Bin Altaf and David A Wood. LogCA: A High-Level Performance Model for Hardware Accelerators. In *ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, pages 375–388, 2017.

[29] Thomas E Anderson, Brian N Bershad, Edward D Lazowska, and Henry M Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 95–109, 1991.

[30] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the Killer Microseconds. *Communications of the ACM*, 60(4):48–54, 2017.

[31] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D Hill, and Michael M Swift. Efficient Virtual Memory for Big Memory Servers. In *ACM International Symposium on Computer Architecture (ISCA)*, pages 237–248, 2013.

[32] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 49–65, 2014.

[33] Jeffrey Dean and Luiz André Barroso. The Tail at Scale. *Communications of the ACM*, 56(2):74–80, 2013.

[34] Dawson R Engler, M Frans Kaashoek, and James O'Toole Jr. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *ACM Symposium on Operating Systems Principles (SOSP)*, page 17, 1995.

[35] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark Silicon and the End of Multicore Scaling. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pages 365–376, 2011.

[36] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 51–66, 2018.

[37] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 281–297, 2020.

[38] Younghwan Go, Muhammad Asim Jamshed, YoungGyoun Moon, Changho Hwang, and KyoungSoo Park. APUNet: Revitalizing GPU as Packet Processing Accelerator. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, volume 17, pages 83–96, 2017.

[39] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. PacketShader: A GPU-accelerated Software Router. In *ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pages 195–206, 2010.

[40] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis. ghost: Fast & Flexible User-Space Delegation of Linux Scheduling. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 588–604, 2021.

[41] Stephen Ibanez, Alex Mallery, Serhat Arslan, Theo Jepsen, Muhammad Shahbaz, Changhoon Kim, and Nick McKeown. The nanoPU: A Nanosecond Network Stack

for Datacenters. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 239–256, 2021.

[42] Zhipeng Jia and Emmett Witchel. Nightcore: Efficient and Scalable Serverless Computing for Latency-Sensitive, Interactive Microservices. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 152–166, 2021.

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

[44] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for μsecond-scale tail latency. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 345–360, 2019.

[45] Kostis Kaffes, Jack Tigar Humphries, David Mazières, and Christos Kozyrakis. Syrup: User-Defined Scheduling Across the Stack. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 605–620, 2021.

[46] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pages 158–169, 2015.

[47] Svilen Kanev, Sam Likun Xi, Gu-Yeon Wei, and David Brooks. Mallacc: Accelerating Memory Allocation. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 33–45, 2017.

[48] Sagar Karandikar, Chris Leary, Chris Kennelly, Jerry Zhao, Dinesh Parimi, Borivoje Nikolic, Krste Asanovic, and Parthasarathy Ranganathan. A Hardware Accelerator for Protocol Buffers. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 462–478, 2021.

[49] Mahmoud Khairy, Ahmad Alawneh, Aaron Barnes, and Timothy G Rogers. SIMR: Single Instruction Multiple Request Processing for Energy-Efficient Data Center Microservices. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 441–463, 2022.

[50] Jongyul Kim, Insu Jang, Waleed Reda, Jaeseong Im, Marco Canini, Dejan Kostić, Youngjin Kwon, Simon Peter, and Emmett Witchel. LineFS: Efficient SmartNIC offload of a distributed file system with pipeline parallelism. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 756–771, 2021.

[51] Nikita Lazarev, Shaojie Xiang, Neil Adit, Zhiru Zhang, and Christina Delimitrou. Dagger: Efficient and Fast RPCs in Cloud Microservices with Near-Memory Reconfigurable NICs. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 36–51, 2021.

[52] Jacob Leverich and Christos Kozyrakis. Reconciling High Server Utilization and Sub-Millisecond Quality-of-Service. In *ACM European Conference on Computer Systems (EuroSys)*, pages 1–14, 2014.

[53] Ao Li, Bojian Zheng, Gennady Pekhimenko, and Fan Long. Automatic Horizontal Fusion for GPU Kernels. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 14–27. IEEE, 2022.

[54] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading Distributed Applications onto SmartNICs using I-Pipe. In *ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pages 318–333. 2019.

[55] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. The Linux Scheduler: A Decade of Wasted Cores. In *ACM European Conference on Computer Systems (EuroSys)*, pages 1–16, 2016.

[56] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: A Microkernel Approach to Host Networking. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 399–413, 2019.

[57] Sarah McClure, Amy Ousterhout, Scott Shenker, and Sylvia Ratnasamy. Efficient Scheduling Policies for Microsecond-Scale Tasks. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 1–18, 2022.

[58] Amirhossein Mirhosseini, Brendan L West, Geoffrey W Blake, and Thomas F Wenisch. Q-zilla: A Scheduling Framework and Core Microarchitecture for Tail-Tolerant Microservices. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 207–219, 2020.

[59] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, volume 19, pages 361–378, 2019.

[60] Simon Peter, Jialin Li, Irene Zhang, Dan RK Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The Operating System is the Control Plane. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–16, 2014.

[61] Arash Pourhabibi, Mark Sutherland, Alexandros Daglis, and Babak Falsafi. Cerebros: Evading the RPC tax in datacenters. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 407–420, 2021.

[62] George Prekas, Marios Kogias, and Edouard Bugnion. Zygos: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 325–341, 2017.

[63] Haoran Qiu, Subho S Banerjee, Saurabh Jha, Zbigniew T Kalbarczyk, and Ravishankar K Iyer. FIRM: An Intelligent Fine-Grained Resource Management Framework for SLO-Oriented Microservices. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[64] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K Aguilera, and Adam Belay. AIFM: High-performance, Application-Integrated Far Memory. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 315–332, 2020.

[65] Rajath Shashidhara, Tim Stamler, Antoine Kaufmann, and Simon Peter. FlexTOE: Flexible TCP Offload with Fine-Grained Parallelism. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 87–102, 2022.

[66] Dharma Shukla, Muthian Sivathanu, Srinidhi Viswanatha, Bhargav Gulavani, Rimma Nehme, Amey Agrawal, Chen Chen, Nipun Kwatra, Ramachandran Ramjee, Pankaj Sharma, Atul Katiyar, Vipul Modi, Vaibhav Sharma, Abhishek Singh, Shreshth Singhal, Kaustubh Welankar, Lu Xun, Ravi Anupindi, Karthik Elangovan, Hasibur Rahman, Zhou Lin, Rahul Seetharaman, Cheng Xu, Eddie Ailijiang, Suresh Krishnappa, and Mark Russinovich. Singularity: Planet-Scale, Preemptive and Elastic Scheduling of AI Workloads, 2022.

[67] Akshitha Sriraman and Abhishek Dhanotia. Accelerometer: Understanding acceleration opportunities for data center overheads at hyperscale. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 733–750, 2020.

[68] Akshitha Sriraman and Thomas F Wenisch. μtune: Auto-tuned threading for OLDI microservices. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 177–194, 2018.

[69] Maroun Tork, Lina Maudlej, and Mark Silberstein. Lynx: A SmartNIC-Driven Accelerator-Centric Architecture for Network Servers. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 117–131, 2020.

[70] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *ACM European Conference on Computer Systems (EuroSys)*, pages 1–17, 2015.

[71] Mario Villamizar, Oscar Garcés, Harold Castro, Mauricio Verano, Lorena Salamanca, Rubby Casallas, and Santiago Gil. Evaluating the Monolithic and the Microservice Architecture Pattern to deploy Web Applications in the Cloud. In *Computing Colombian Conference*, pages 583–590, 2015.

[72] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S. Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, Pedro Henrique Penna, Max Demoulin, Piali Choudhury, and Anirudh Badam. The Demikernel Datapath OS Architecture for Microsecond-Scale Datacenter Systems. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 195–211, 2021.

[73] Yanqi Zhang, Weizhe Hua, Zhuangzhuang Zhou, G Edward Suh, and Christina Delimitrou. Sinan: ML-based and QoS-aware resource management for cloud microservices. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 167–181, 2021.