Parallelism in Network Processing

Tom Herbert
SiPanda

ABSTRACT

Network processing is inherently a form of serialized data processing where packets, as well as protocol layers in packets, must be processed in sequential order without overlap. The ever increasing demands for low latency and high throughput motivate the pursuit of higher performance in network processing. While parallelism is a well known technique in computing for high performance, parallelizing network processing with granularity has been challenging. In this paper, we present a novel solution that reaps the maximum benefits of parallelism in network processing. We first provide a theoretical foundation that defines horizontal and vertical parallelism as the fundamental types of parallelism in network processing. We then describe a threading and synchronization model designed around the characteristics of parallelizing serial data processing. Given this foundation, we describe the Serial Data Processing Unit, a domain specific architecture for fine grained and easily programmable parallelism in network processing. Lastly, we describe the Single Flow Bottleneck and how the techniques outlined in this paper can address it. The benefits of our solution are higher throughput, lower latency, and better resource utilization.

CCS CONCEPTS

• **Computer systems organization → Architectures;Parallel architectures;Multicore architectures**

• **Networks → Network algorithms;Data path algorithms**

KEYWORDS

Parallelism, Synchronization, Parsing, Data path, Serial data, Parallel programming. Programmable Dataplane

1. INTRODUCTION

In this paper we introduce a foundation and architecture for parallelism in network processing[1]. Network processing is inherently a form of serial data processing, hence our goal is to "find" parallelism in serial data processing [55]. Heretofore, this has proven to be a challenging endeavor.

State of the art of parallelism in network processing is multi-queue. While multi-queue has served us well over the years, its limitations and edge conditions are a mismatch for the demands for higher performance. This is especially true as we're reaching the end of Moore's Law [34,48] and we can no longer rely on increases in CPU speed to keep pace with increases in network bandwidth. More parallelism is one answer to the end of Moore's Law [33].

Our architecture exploits key attributes of serial data processing to elicit fine grained parallelism in network processing. This is based on the observation that per the semantics of serial data processing, the externally visible output of network processing must correspond to both packets and their protocol layers having been processed in order. We implement an internal parallelism in network processing that properly maintains the effects of serial data processing. In concert with other techniques, including hardware accelerators, memory optimizations, and ISA extensions; parallelism can greatly improve performance in latency, throughput, and resource utilization.

This is a "Systems Paper" focused on scaling networking performance of data center servers for processing very high packet rates (up to one billion packets per second). In our performance analysis, we assume that server CPU processing is the primary bottleneck for processing high packet rates, and that the network and other system resources can be sufficiently provisioned to avoid being bottlenecks.

This paper is organized as follows. First we describe multi-queue and its problems. Then, we provide the theoretical foundation for parallelism in network processing, and we describe our threading and synchronization model. We then present the *Serial Data Processing Unit*, a domain specific architecture [22] for network processing. Next, we describe the *Single Flow Bottleneck* and demonstrate how the techniques of this paper can be applied to address it. We conclude with discussion and opportunities.

[1] Original inventions in this paper are covered by US patent

2. MULTI-QUEUE

Multi-queue parallelism, or *multi-queue* [12], is commonly supported in host networking implementations. Multi-queue steers received packets to different queues, and each queue is processed by a CPU thread where the threads can run in parallel[2]. The in-order processing requirements are relaxed so that only packets for the same flow must be processed in order. Packets are steered to queues based on a classification algorithm, such as a hash over the transport layer 4-tuple, that associates packets with their flows. This creates affinity between flows and threads to ensure that packets in a flow are processed in order, but it also promotes cache locality of flow related data structures. Figure 1 illustrates multi-queue.

Multi-queue is an opportunistic optimization. Under ideal conditions, there is a mix of packets in many different flows such that the load across receive queues has a uniform distribution (Figure 2(a)); we call this *idealized parallelism* and an arbitrarily high throughput can be achieved given enough queues and threads. Under less than perfect conditions, multi-queue has two fundamental problems: 1) workload imbalances, and 2) the Single Flow Bottleneck. The performance effects of these problems are shown in Figure 3.

2.1 Multi-queue workload imbalances

Multi-queue is susceptible to workload imbalances. The workload across receive queues can be nonuniform when there is a relatively small number of active flows[3]. This is due to variance of distribution for the hash function which is only asymptotic to a uniform distribution.

A proposed mitigation to the workload imbalance problem is to steer packets for flows to queues by round robin; for example, packets of the first flow are steered to the first queue, packets of the second flow are steered to the second queue, and so on. This produces a uniform distribution for good performance, however steering packets by round robin requires state that maps flows to queues, so it is less practical to implement than hash steering which is stateless.

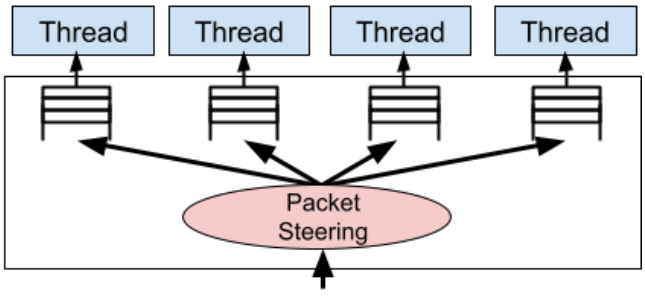

Figure 1: Multi-queue parallelism. Packets are received on a network interface and are "steered to queues" based on their flow.

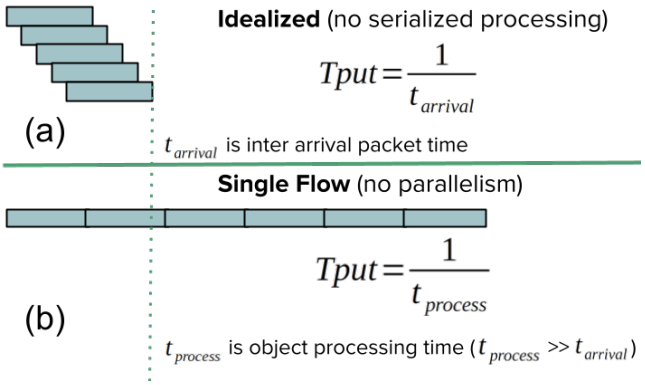

Figure 2: Best and worst case scenarios of multi-queue performance. (a) illustrates idealized parallelism in which all received packets are independent and can be processed in parallel. (b) shows the worst case scenario in which all packets are for a single flow and are processed by one thread with no parallelism. Each block represents a packet, and the dotted line indicates line rate throughput. *Tput* gives the throughput for the scenarios.

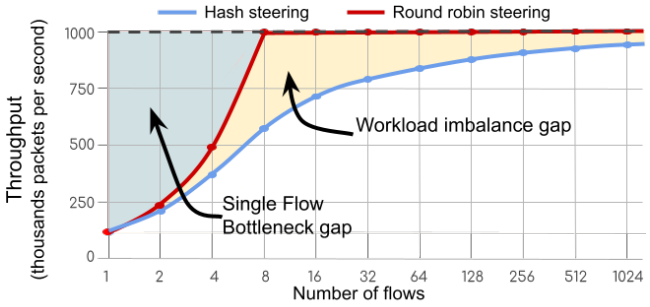

Figure 3: Multi-queue performance. In this example, eight receive queues are used, and 125,000 packets per second can be processed by each queue for a maximum throughput of one million packets per second. The light blue line shows the performance of multi-queue when steering by hash, and the dark red line shows performance of multi-queue when steering with round robin. The dashed line indicates maximum throughput of one million packets per second which is attainable with horizontal parallelism (Section 3.1). The performance gaps from workload imbalances and the Single Flow Bottleneck are highlighted.

2.2 Multi-queue Single Flow Bottleneck

If the number of active flows is less than the number of queues, then an imbalanced workload is inevitable with multi-queue even with round robin steering. For instance, if there are eight queues and only four flows, all packets for those four flows are processed by at most four CPUs which limits throughput. In the worst case, there is only one active flow so that all packets are processed by one CPU with no parallelism (Figure 2(b)). We call this problem the *Single Flow Bottleneck*. Section 7 describes the problem in detail and presents a solution based on the techniques and architecture described in this paper.

[2] There are software and hardware variants of multi-queue [25]. Receive Side Scaling (RSS) [59] is a hardware technique commonly implemented in NICs (Network Interface Cards) that steers packets to different hardware queues based on a hash of the flow tuple in packets. Receive Packet Steering (RPS) emulates RSS in software. Receive Flow Steering (RFS) is an extension of RPS that steers packets to CPUs where the associated flow is being processed. Accelerated RFS (aRFS) implements a form of RFS in NIC hardware.

[3] This problem has been exploited in a Denial of Service attack where an attacker saturates a queue with packets in an attempt to degrade service for other flows mapped to the same receive queue [13].

3. FOUNDATIONS OF PARALLELISM

In this section, we provide the theoretical foundation for parallelism in network processing, and describe the two fundamental types of parallelism in network processing: horizontal parallelism and vertical parallelism[4]. Figure 4 illustrates these.

3.1 Horizontal parallelism

In *horizontal parallelism*, packets are processed in parallel in different threads[5]. Packets are steered similarly as in multi-queue but without flow to CPU affinity which avoids workload imbalances. Synchronization (Section 4) is used to preserve correct ordering. An example of horizontal parallelism is a network switch that processes packets in parallel but forwards them in order.

The minimal number of parallel threads required to handle packet processing at maximum input rate in horizontal parallelism is given by Equation 1[6].

$$numthreads = ceiling\left(\frac{t_{packet}}{t_{input}}\right)$$

Equation 1: Minimal number of threads to sustain a given input rate in horizontal parallelism. t_{input} is the minimum time between consecutive packets, where the inverse of t_{input} is the input rate. t_{packet} is the maximum time to process a packet.

The number and types of protocol layers in packets can vary, so the total time to process a packet in horizontal parallelism is the sum of times to process each protocol layer of the packet. This is given by Equation 2.

$$t_{packet} = \sum_{i=all\,layers} t(i)_{layer}$$

Equation 2: Maximum time to process a packet in horizontal parallelism. $t(i)_{layer}$ is the maximum time to process the i'th protocol layer in a packet.

The time to process one protocol layer of a packet in a CPU is a function of the number of instructions, Instructions Per Cycle (IPC), and CPU clock frequency[7] [21]. This function is given in Equation 3.

$$t_{layer} = \left(\frac{NumIns_{layer}}{IPC}\right) \times \left(\frac{1}{ClockFreq}\right)$$

Equation 3: Time to process one protocol layer of a packet in a CPU. $NumIns_{layer}$ is the number of instructions executed for processing the protocol layer , *IPC* is the Instruction Per Cycle, and *ClockFreq* is the CPU clock frequency.

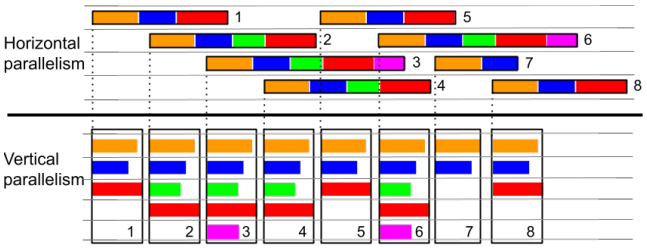

Figure 4: Horizontal and Vertical parallelism. The diagram illustrates an example for processing eight packets in both horizontal and vertical parallelism. Each packet is composed of some number of protocol layers (represented by the colored blocks). In this example, packets are input at constant intervals as indicated by the dotted vertical lines. In horizontal parallelism the packets are processed in parallel in four threads, and in vertical parallelism the individual protocol layers of each packet are processed in parallel using five threads. Each marked row in the diagram represents a thread.

Multi-queue is essentially a constrained form of horizontal parallelism. The primary difference between the two is that horizontal parallelism steers packets without regard to their flow; this allows packets to be "randomly sprayed" across threads which guarantees a uniform distribution and high throughput regardless of the number of flows (Figure 3). Horizontal parallelism doesn't establish flow to CPU affinity, so it doesn't promote cache locality of flow related data structures as multi-queue does. Section 7 discusses mitigations for the loss of cache locality.

3.2 Vertical parallelism

In *vertical parallelism*, protocol layers of a single packet are processed in parallel in different threads[8]. Per the requirements of serial data processing, the protocol layers must be processed in order. Synchronization, described in Section 4, is used to preserve correct ordering. An example of vertical parallelism is a server that processes TCP/IPv6 packets where each of the Ethernet, IPv6, and TCP headers are processed in parallel in different threads. The primary benefit of vertical parallelism is reduced per packet latency.

In vertical parallelism, the total time to process a packet equals the maximum time to process any of the constituent protocol layers. This is given by Equation 4.

$$t_{packet} = \max_{i=all\,layers} \left(t(i)_{layer}\right)$$

Equation 4: Maximum time to process a packet in vertical parallelism. $t(i)_{layer}$ is the maximum time to process the i'th protocol layer in a packet.

---

[4] The term "horizontal" is inspired by envisioning serial data as a continuous bit stream, like the memory tape of a Turing machine [58]; the term "vertical" is inspired by the typical vertical visualization of protocol stacks, like how the OSI model is visualized [38].

[5] Architecturally, horizontal parallelism is a form of MIMD (Multiple Instructions, Multiple Data) parallelism [16].

[6] In Equation 1, if t_{packet} is less than t_{input} then one thread is sufficient to handle the workload which is desirable in some use cases.

[7] A clock cycle, or just *cycle*, is a discrete time event in a CPU that drives instruction execution. The CPU clock frequency is the number of cycles per second. The number of instructions executed per clock cycle, or IPC, is dependent on the CPU architecture and workload [21]. We measured the IPC on x86 for network processing to be in the range of 1.4 to 1.8. For our purposes, we assume an average IPC of 1.4.

[8] Architecturally, vertical parallelism is a form of MISD (Multiple Instructions, Single Data) parallelism [16], where the "Single Data" is one packet. It may be arguable if vertical parallelism corresponds to a strict definition of MISD as a processor architecture, however from a system level view MISD seems to be the best classification and differentiates from horizontal parallelism which is clearly MIMD.

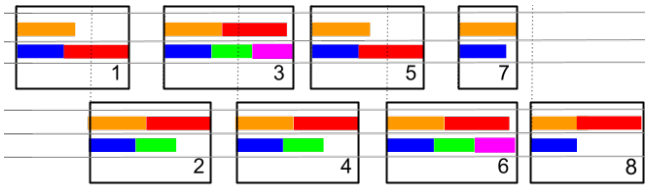

Figure 5: Hybrid and constrained vertical parallelism. The diagram illustrates an example for processing the same eight packets in Figure 4 in hybrid parallelism with constrained vertical parallelism. Two packets can be processed in horizontal parallelism, and two protocol layers of each packet can be processed in parallel in vertical parallelism. If a packet has more than two protocol layers then processing has to wait for an available thread. For instance, in packet #2, processing for the red protocol layer waits until the orange layer processing completes.

3.3 Constrained Vertical Parallelism

If the number of protocol layers in a packet is greater than the number of threads available for vertical parallelism then that is *constrained vertical parallelism*. In constrained vertical parallelism, if no threads are available then the processing for a protocol layer must wait until one is available. A thread becomes available when processing for another protocol layer completes. In this architecture, packet processing is work conserving [29], so that there is always at least one thread running and guaranteed to complete. Figure 5 demonstrates an example processing flow with constrained vertical parallelism.

For constrained vertical parallelism, the time to process a packet is given by Equation 5.

$$t_{packet} = ceiling\left(\frac{numlayers}{numthreads}\right) \times \max_{all\ layers}\left(t(i)_{layer}\right)$$

Equation 5: Maximum time to process a packet in constrained vertical parallelism. $t(i)_{layer}$ is the maximum time to process the i'th protocol layer, *numthreads* indicates the number of threads available for vertical parallelism, and *numlayers* indicates the number of protocol layers in a packet. Note that if *numthreads* >= *numlayers* then the equation is equivalent to Equation 4 for vertical parallelism, and if *numthreads* is one then the equation degenerates to Equation 2 with no vertical parallelism.

3.4 Hybrid parallelism

To achieve high throughput and low latency, a combination of vertical parallelism and horizontal parallelism may be employed in *hybrid parallelism*. Figure 5 demonstrates an example processing flow using hybrid parallelism.

Per Equation 1, hybrid parallelism becomes effective when the packet processing time is greater than inter packet arrival time and is especially beneficial for high throughput cases, such as a high speed network switch, where per packet processing time is much greater, possibly by one or two orders of magnitude, than inter packet arrival time Vertical parallelism bounds the latency to process a single packet which minimizes packet processing time and the number of horizontal threads needed.

4. THREADING

This section describes the threading and scheduling model for horizontal and vertical parallelism. The elements of this model are threads, thread sets, and datapaths; these are illustrated in Figure 6.

4.1 Threads

A *thread* is the smallest set of programmed instructions that can be independently scheduled to execute in a CPU [44]. Threads are supported by the underlying operating system to provide *thread level parallelism*[9] [45]. Threads are run to completion [52] and should have minimal context switch overhead [53] to support vertical parallelism.

A number of OS level threads are created to perform network processing. Each thread runs an event loop that polls a work queue for new work to be performed. A queue contains *work items*, each of which describes a unit of work being requested. A work item includes all the necessary information for processing a layer including a reference to the packet being processed, the specific function that the thread should perform, and other contextual information needed for processing. When work becomes available to a thread, meaning there is an item in the work queue, it dequeues the first item in the queue and performs the requested processing by calling the indicated function. When the processing function returns, the thread is *done* and the event loop is re-initialized to process the next work item. A thread will block while waiting on a dependency to be resolved (Section 5) or waiting for an accelerator to complete. While a thread is blocked, another thread can be scheduled to run on the CPU.

4.2 Thread sets

Threads are grouped together into *thread sets*. A thread set defines one instance of packet processing, and thread sets can process packets in parallel in horizontal parallelism. A thread set contains a number of *worker threads* that process protocol layers of packets in vertical parallelism. At any given time, a worker thread is either *available* (not running), or *busy* (running and processing a protocol layer).

The set of running threads in a thread set are well ordered and maintained by the thread set in an *ordered list.* The ordered list determines the downstream and upstream relationships of threads needed for dependency resolution (Section 5). When a thread set is started, it is added to the tail of the ordered list; and when a thread terminates (becomes "done"), it is removed from the ordered list.

Threads may *kill* downstream threads if a condition renders work of the downstream threads irrelevant. For instance, if IPv4 header checksum validation fails when processing an IP header then the packet will be dropped and processing of the TCP header is no longer relevant, so the IP layer processing thread can kill the TCP processing thread along with any downstream threads of the TCP processing thread.

[9] POSIX Threads (*pthreads*) [32] provides this functionality. To reduce overhead, we are investigating the use of *fibers* [36]. In the SDPU, hardware threads are used and are managed by a hardware CPU scheduler for very low overhead.

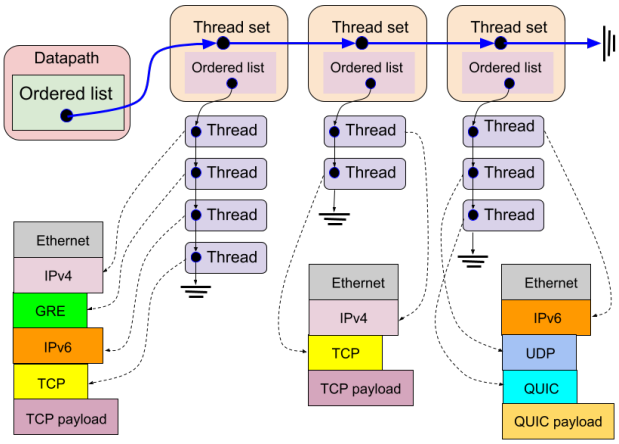

Figure 6: Threads, thread sets, and datapaths. This example shows a datapath with three thread sets processing packets. Threads within each thread set are scheduled to process protocol layers of the packet being processed by the thread set. Note that thread sets are in the ordered list of the datapath, and threads are in the ordered list of their thread set.

4.3 Datapaths

A *datapath* is an instance of a network processing data path, and consists of a number of thread sets. Thread sets of a datapath can process packets in parallel in horizontal parallelism. Thread sets are well ordered and maintained in an ordered list. The ordered list determines the downstream and upstream relationships of thread sets needed for inter thread set dependency resolution (Section 5.4). When a thread set is started it is added to the tail of the ordered list; and when all the threads in a thread set are done then the thread set is *done* and it is removed from the ordered list.

4.4 Worker thread scheduling

Worker threads are scheduled by *top function scheduling* or *cascade scheduling*. These can be combined in *hybrid scheduling*. Figure 7 illustrates these methods.

4.4.1 Top function scheduling

In *top function scheduling*, each thread set runs a *top function* to schedule worker threads. Typically, the top function invokes a parser [17] to identify the constituent protocol layers of packets and then schedules threads to process the layers. For each identified protocol layer, the top function may schedule a worker thread by queuing a work item on the work queue of an available worker thread. Parsing may be tightly integrated with scheduling as *parser-scheduling*, and the top function is then a *parser-scheduler*.

4.4.2 Cascade scheduling

In *cascade scheduling*, the last worker thread in the ordered list can schedule the next thread in a thread set. When the first thread runs it can schedule a second worker thread, the second thread may in turn schedule a third worker thread, and so on. The cascade of scheduling threads stops when a last worker thread doesn't schedule a next thread. In cascading scheduling, *only the last thread in the thread set's order list may schedule a next thread*.

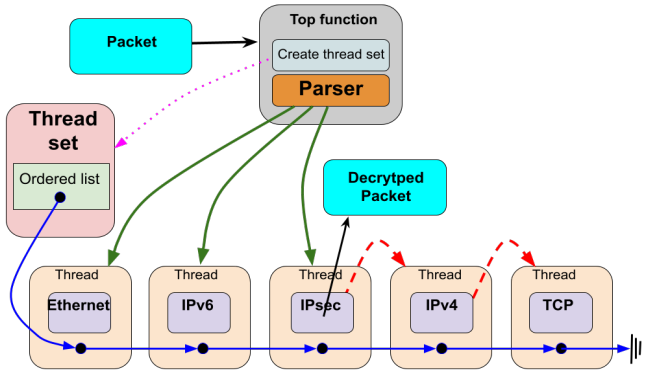

Figure 7: Example of hybrid scheduling with top-function (parser-scheduling) and cascade scheduling. In this example, hybrid scheduling is used to schedule worker threads to process a TCP in IPv4 in IPsec in IPv6 in Ethernet packet; IPsec encrypts the encapsulated IPv4 packet. When a packet is input, the parser-scheduler parses the Ethernet, IPv6, and IPsec headers (those in plain text), and schedules worker threads to process them as indicated by the solid green arrows from the top function. IPsec processing decrypts the encapsulated IPv4 packet and schedules a thread to process the IPv4 header in cascade scheduling. The IPv4 thread then schedules a thread to process the TCP header. Cascade scheduling is indicated by the dashed red arrows.

4.4.3 Hybrid scheduling

Top function scheduling and cascade scheduling can be used in tandem in *hybrid scheduling*. In hybrid scheduling, the top function schedules some number of threads via top function scheduling. The last thread started by the top function may schedule the next thread to initiate a cascade. In hybrid scheduling, *cascade scheduling may commence only after the top function has scheduled all its threads*

4.5 Closed thread sets

Once the last and final thread has been scheduled in a thread set, the thread set is *closed*. When a thread set is in closed state, no additional worker threads can be started and no more threads are added to the ordered list. Closing a thread set initiates propagation of resolved dependencies between threads sets as described in Section 5.4.

5. SYNCHRONIZATION

Synchronization [19,46] is the coordination of concurrent threads of execution to ensure that they don't interfere with each other or access shared resources in an inconsistent or unsafe way. For instance, when two threads are processing received packets for the same TCP connection in horizontal parallelism, updates to the Protocol Control Block (PCB) [51] must be synchronized to ensure correctness. Similarly, if the IP and TCP protocol layers of a TCP/IP packet are processed in parallel, TCP processing cannot update the PCB until the IP processing completes and has verified the packet is correct. Historically, locks or mutexes [18] have been used for synchronization; however, in this paper we introduce a novel mechanism called *dependencies* that is specifically designed for synchronization in serial data processing. The key design point is that one thread can have a *dependency* on processing in another thread .

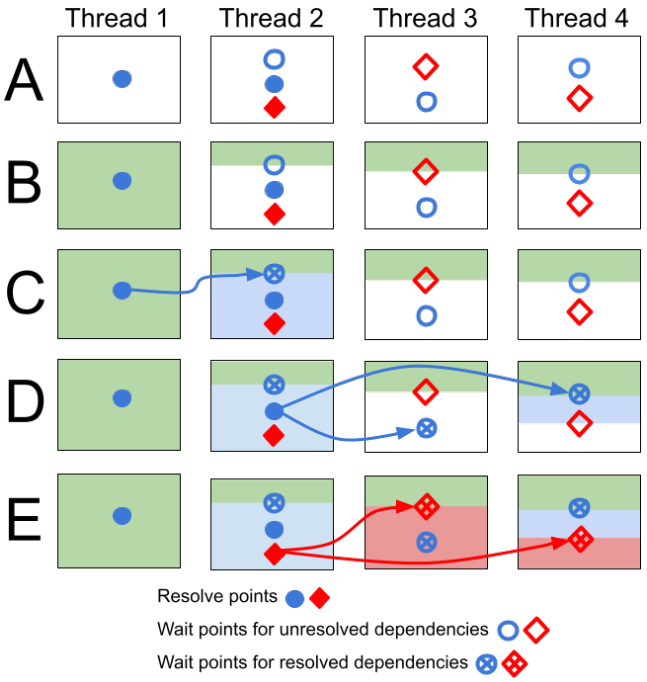

Figure 8: Example of dependency resolution. The columns, labeled Thread 1 to Thread 4, refer to four threads in a thread set. The rows of the diagram, labeled A to E, provide points in the timeline for discussion. There are two dependencies: the "circle dependency" and the "diamond dependency".

Line A shows the state of dependencies before execution commences. Line B shows the initial execution allowed in the four stages-- green areas represent portions of the code path that have no dependencies and can run in parallel from the start. In Line C, Thread 1 resolves the circle dependency-- at this point the blue section in Thread 2 can now run. In Line D, Thread 2 resolves the second instance of the circle dependency-- at this point the blue section in Thread 4 can run; although the circle dependency is now resolved for Thread 3, it cannot proceed since it is still waiting on the diamond dependency. Finally, in Line E Thread 2 resolves the diamond dependency so that the red portions of Thread 3 and Thread 4 can now run.

A thread with a dependency cannot proceed to execute in the critical region corresponding to the dependency until the thread it depends on *resolves* the dependency. Dependencies are strictly unidirectional so that a thread can have a dependency on an upstream thread, but not the other way around. This unidirectional property ensures that there are no circular dependencies, and hence deadlock [47] with dependencies is not possible.

5.1 Wait and resolve points

Dependency synchronization is expressed as resolve points and wait points. A *resolve point* is a point in the code path of a thread at which processing has been done to satisfy a dependency in downstream threads. A *wait point* is a point in the code path of a thread at which execution cannot proceed until the dependency has been resolved by upstream threads. *Wait* and *resolve* primitives are defined in an API to synchronize between wait and resolve points.

As an example, consider a dependency between threads processing the TCP and IP layers of a packet. The TCP thread can perform basic validation of the TCP header, such as checksum validation, in parallel with IP layer processing. However, the TCP thread cannot commit changes to the TCP PCB before the IP thread has validated the IP header. An "IP layer accepted packet" dependency may be employed with a wait point in the TCP layer code path before updating the PCB, and a resolve point in the IP layer code path after the packet is validated.

5.2 Dependency resolution signals

When a thread resolves a dependency, downstream threads in the thread set are informed of the dependency resolution and execution can proceed through wait points for the dependency. This is done by propagating a *resolution signal* to downstream threads[10]. A resolution signal propagates until one of the following conditions are met:

1. The end of the ordered list for the thread set is reached.
2. A thread is encountered that blocks, that is it must resolve, the same dependency being resolved.
3. A thread is encountered that has already resolved the same dependency.

Figure 8 shows an example of dependency resolution.

5.3 Watchers, blockers, and waiters

With respect to a dependency, a thread may be a watcher, blocker, or waiter. A *dependency watcher* is a thread that is interested in a dependency and may wait on it. A *dependency waiter* is a thread actively waiting on a dependency to be resolved; while it's waiting the thread is blocked and cannot proceed until the dependency is resolved. A *dependency blocker* is a thread that blocks a dependency; the thread must resolve the dependency before the resolution signal is propagated to downstream threads. When a thread starts, the dependencies that the thread blocks are declared and the thread set maintains a list of blockers for each dependency. When a thread terminates, any unresolved blocked dependencies are automatically resolved. A thread may be a blocker, watcher and waiter of the same dependency. Figure 9 illustrates the operation of watchers, blockers, and waiters.

5.4 Inter thread set dependencies

Dependency resolution may be propagated from one thread set to another in the ordered list of thread sets of a datapath. Dependencies that can be propagated between thread sets are called *propagated dependencies* and are declared in the configuration of a datapath. Propagated dependencies must be a subset of all the dependencies defined for a datapath, and a dependency is either propagated or non-propagated. With regards to dependency resolution amongst threads in a thread set, propagated dependencies are indistinguishable from non-propagated dependencies.

[10] In an OS, the resolution signal could be implemented using a condition variable [2].

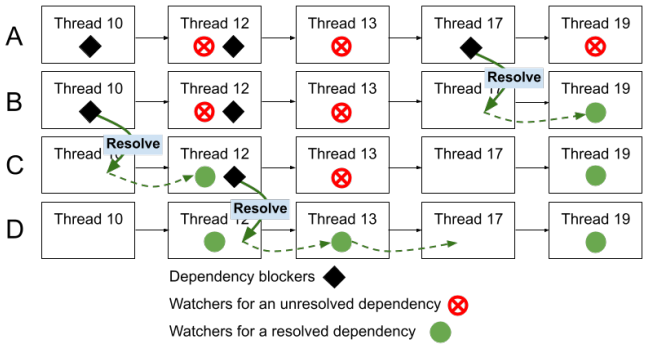

Figure 9: Example of dependency watchers, blockers, and waiters: The diagram shows the list of blocker and watcher threads for a dependency. Once a blocker has resolved a dependency, it is removed from the list of blockers for the dependency. A watcher becomes a waiter when it is actively waiting on a dependency (waiters are not shown). The rows, labeled A to D, provide points in the timeline for discussion.

In the initial state, Line A, there are three blockers and three watchers: Thread 10 and Thread 17 are blockers but not watchers, Thread 13 and Thread 19 are watchers but not blockers, and Thread 12 is both a watcher and a blocker. In Line B, Thread 17 resolves the dependency. The resolution signal is propagated to Thread 19 and then stops since the end of the ordered list is reached. Note that Thread 17 is not a watcher so that it is effectively creating a new independent instance of the dependency In Line C, Thread 10 resolves the dependency. The resolution signal is propagated to Thread 12, but stops since Thread 12 is a blocker. Subsequently in Line D, Thread 12 resolves the dependency. The resolution signal is propagated to Threads 13 and 17, but stops there since the dependency is already resolved for Thread 19.

Propagated dependencies that have been resolved and are not blocked by the last thread of a closed thread set may be resolved in the next thread set in the datapath's ordered list. Resolving a dependency in the next thread set is done by propagating a dependency resolution signal starting from the first thread in the next thread set.

If a dependency resolution is propagated between thread sets and the resolution signal reaches the last thread in the following thread set which is also closed and doesn't block the dependency, then the dependency resolution may be further propagated to the "next next" thread set.

5.5 Dependency channels

In-order processing requirements may be relaxed such that packets in the same flow are processed in order, but packets in different flows have no processing order requirements (similar to multi-queue). Thread sets processing packets of the same flow are grouped together in a *dependency channel*. Each dependency channel contains an ordered list of thread sets, and in-order processing is maintained within the channel. A datapath maintains a table of dependency channels. Figure 10 illustrates dependency channels.

A thread set *joins* a dependency channel via a "join channel" operation. A thread set is joined to a channel by adding it to the tail of the ordered list of thread sets for the channel. Once joined to a dependency channel, a thread set

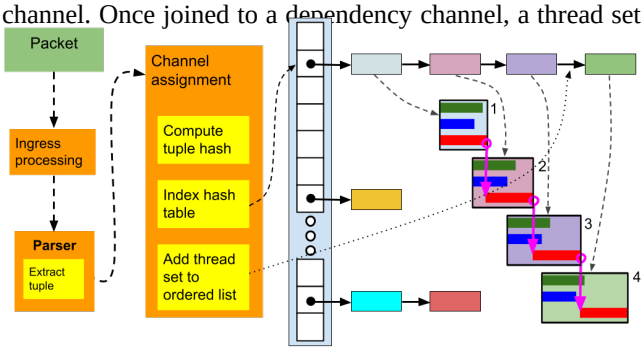

Figure 10: Example of channel dependencies. Dependency channels are organized in a hash table, where each element contains an ordered list of thread sets joined to a channel. When a packet is received it is joined to a channel by hashing over the identifying 4-tuple of the packet, using the result to index the hash table, and then adding the thread set to the tail of the ordered list for the dependency channel. In the diagram, the thread sets for the second channel are expanded to show their threads and dependency resolution propagation for a channel dependency.

is joined for its remaining lifetime processing a packet. To avoid deadlock, *the relative ordering of thread sets in a dependency channel must be the same as that in the ordered list of the datapath*. When a thread set is done and is joined to a dependency channel, it is removed from the ordered list of the dependency channel.

When a thread set starts, it is not automatically joined to a channel. Some processing is needed on the packet to determine its flow, and hence which channel to join[11]. A thread set is not required to join a channel, as might be the case of a packet with no associated flow.

5.6 Channel dependencies

A set of *channel dependencies* is used for synchronization amongst the thread sets of a channel. Channel dependencies are declared in datapath configuration and are a subset of propagated dependencies for a datapath. A propagated dependency is either a "channel dependency" or a "non-channel dependency".

Channel dependencies that have been resolved and are not blocked by the last thread in a closed thread set can be propagated to the next thread set in the ordered list of the dependency channel to which the thread set joined. The channel dependencies are used across all channels in a datapath, but they are only propagated amongst thread sets of the same channel and so are effectively independent sets of dependencies. Channel dependencies are propagated between thread sets of a dependency channel following the rules for inter thread set dependencies as described in Section 5.4.

[11] Typically, the identifying flow tuple in packets is hashed to determine a flow. For instance, for a TCP/IP packet a hash might be computed over the TCP 4-tuple which includes the IP addresses and TCP port numbers. Commonly used hash functions are Jenkins [27], Toeplitz [28], and SipHash [5].

6. SERIAL DATA PROCESSING UNIT

The *Serial Data Processing Unit*, or *SDPU*, is a domain specific processor and architecture for fully programmable, network processing and parallelism. The architecture of the SDPU is shown in Figure 11. Figure 12 shows the flow of a packet through the SDPU and the relationships between components.

The SDPU consists of clusters of RISC-V [4] CPUs, where each cluster contains some number of worker CPUs and parser CPUs. *Worker CPUs* provide hardware threads to run the threads described in Section 4.1. *Parser CPUs* work in tandem with *cluster schedulers* to perform parser-scheduling (Section 4.4.1) of threads that run in worker CPUs. Worker and parser CPUs are user programmable.

Outside of the CPU clusters are the dispatcher, global scheduler, and accelerators. The *dispatcher* dispatches received packets to clusters to be processed. The *global scheduler* maintains the ordered list of thread sets and dependency channels for a datapath. *Accelerators* accelerate functions on behalf of worker CPUs. These components are generally not user programmable and implement the "uncore" functions in the SDPU.

6.1 Implementation

We implemented a Proof of Concept (PoC) of the SDPU in a software emulator and in FPGA.

The software emulator runs on a Linux RISC-V virtual machine (VM) in either Docker [43] or QEMU [1]. The emulator is a fully functional implementation of the SDPU, and is provided as a library to be linked with test programs. The emulator is configurable to allow quick evaluation of different configurations for the numbers of clusters, worker CPUs, parser CPUs, and accelerators. It also allows us to extrapolate performance of hardware implementation.

For the FPGA implementation we run ten RISC-V CPUs in a Xilinx U200 FPGA [61]. One cluster is created with two worker CPUs and a parser CPU. The rest of the CPUs are dedicated to emulating the uncore components of the SDPU[12]. We also implemented some accelerators in RTL[13]. Communication amongst components is message based and uses hardware FIFOs for high performance[14]. The FPGA implementation allows us to test with native RISC-V CPUs and to develop uncore components in RTL [8].

Users write applications for their data path in the PANDA programming model [39]. A program includes the code for parser CPUs and worker CPUs. We modified the LLVM compiler [31] to optimize compilation for the SDPU CPUs, including emitting domain specific custom instructions supported in parser and worker RISC-V CPUs.

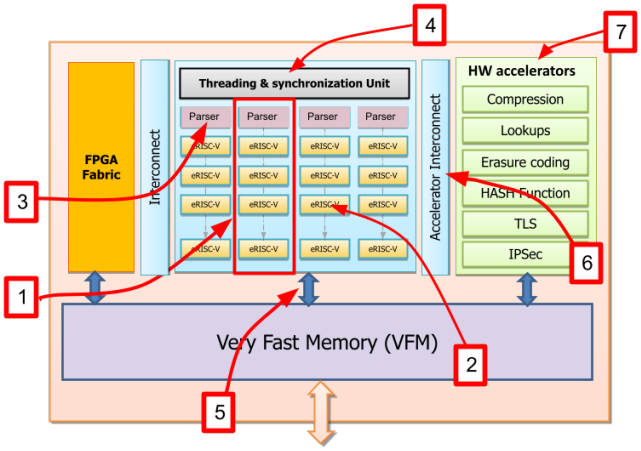

Figure 11: Annotated block diagram of the Serial Data Processing Unit (SDPU). The components of the SDPU include: (1) some number of CPU clusters, (2) each cluster has some number of enhanced RISC-V worker CPUs, (3) each cluster has at least one RISC-V parser CPU, (4) a threading and synchronization unit that includes the cluster schedulers and the global scheduler that manage scheduling and dependencies for horizontal and vertical parallelism, (5) a high performance interface to very fast memory (6), a high performance interconnect fabric to accelerators, (7) external accelerators (in ASIC, FPGA, etc.).

6.2 SDPU Performance

The SDPU is designed to maximize throughput and minimize latency. Throughput can be measured by packets processed per second (pps). Latency is measured as the time from when a packet starts to be processed to the time that processing completes. Tail latency is a key metric [14], so we consider the 99.9th percentile of tail latency.

Parallelism is an important technique for high throughput, and is employed in the SDPU at multiple levels. The major components operate in parallel as a pipeline. Thread sets are instantiated in clusters for horizontal parallelism, and worker threads run in worker CPUs for vertical parallelism. A cluster may have multiple parser CPUs that can run in parallel. Accelerators can run in parallel with CPU processing and other accelerators, and they may also employ internal parallelism.

Latency is a function of the amount of serialized execution in processing a packet, so we use vertical parallelism to reduce the time spent in serialized processing. Tail latency correlates to variances in the processing path. Variance is minimized in the SDPU by deterministic processing-- no OS, no interrupts, no cache misses, no exceptions, etc.

In this section, we analyze performance of workloads that exhibit idealized parallelism where latency and throughput are independent; we use a TCP SYN cookies program as a reference application. In Section 7 we consider workloads where processing latency correlates with throughput.

[12] The final design goal is for all the uncore components in the SDPU to run in RTL. Worker and parser CPUs are intended to run as bare metal CPUs without an OS and with a hardware CPU scheduler that orchestrates processing threads.

[13] To date, we implemented SipHash [5], CRC [40], Murmur3 hash [3], and LZF compression [62] as hardware accelerators in RTL.

[14] Hardware FIFOs are lockless, and enqueue and dequeue operations are respectively done by a single memory store and load.

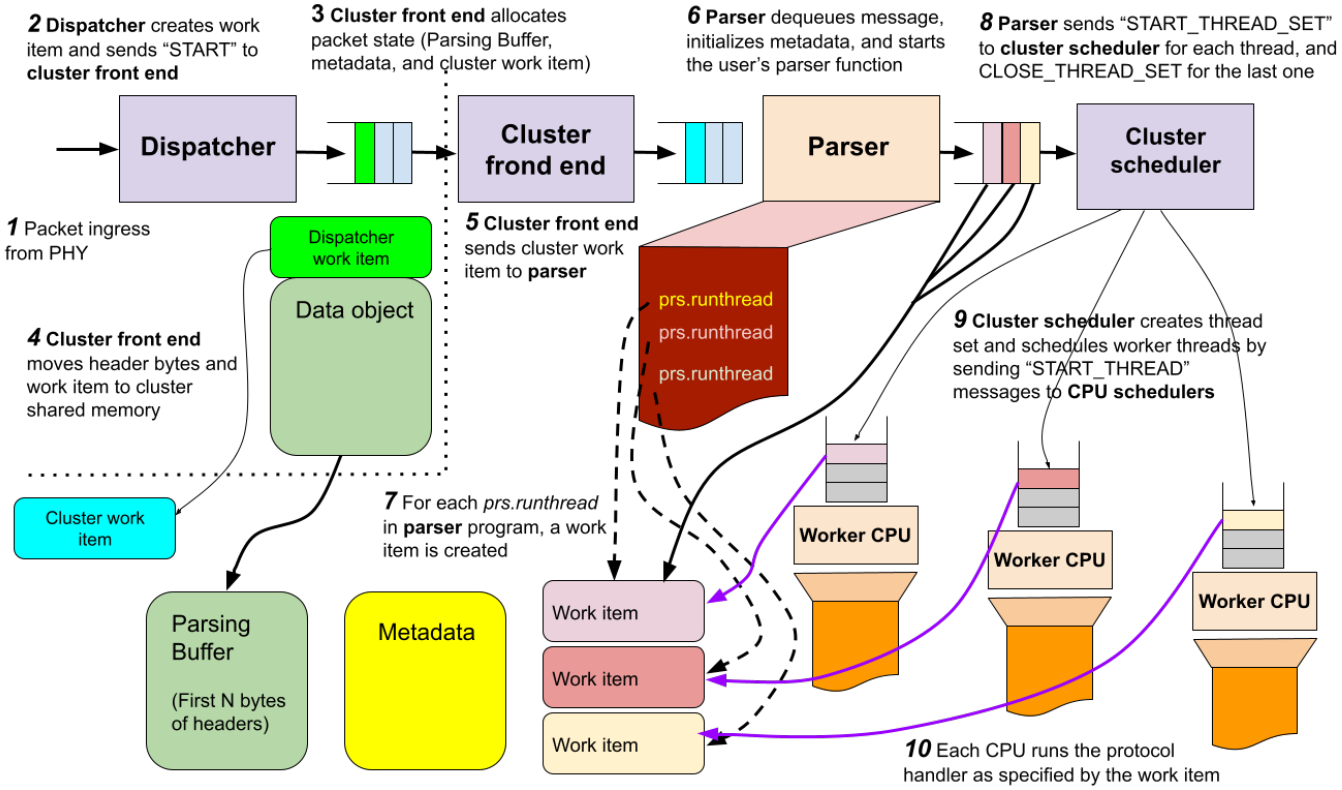

Figure 12: SDPU processing flow. The steps for processing a packet are numbered 1 to 10. Communication between components is performed through message FIFOs. For efficiency, the SDPU operates on a parsing buffer containing the first N bytes of headers (parsing buffers are a common technique in high performance routers). Parser and worker CPUs run user written programs.

6.2.1 Reference program for performance analysis

To evaluate performance, we wrote a program for the SDPU and Linux to generate TCP SYN cookies [11]. This is a stateless application that receives TCP SYN packets, and then parses and validates the Ethernet, IPv4 headers, TCP headers, and TCP options. If the validations succeed for a SYN packet then the program creates a TCP SYN-ACK packet with a SYN cookie. Table 1 shows the number of assembly instructions for the two implementations.

6.2.2 Worker CPU Performance

Worker CPUs are optimized so that 99% of CPU cycles are dedicated to executing instructions in user programs without pipeline stalls. This is accomplished by:

- Eliminating infrastructure overhead (no OS)
- Accelerating complex functions like crypto
- Avoiding cache misses, prefetching as needed
- ISA extensions (e.g. checksum calculation[15])
- Low latency thread context switch

With these optimizations there are no wasted CPU cycles and processing is deterministic so that mean latency and tail latency are minimized. Assuming idealized parallelism, the number of CPUs needed to achieve a given throughput is a straightforward calculation using Equation 1.

Table 1: Number of instructions to generate TCP SYN cookies. This compares the number of instructions needed in Linux on x86 and in the SDPU to process the fields of a TCP-SYN packet and generate a SYN cookie. Note the SDPU divides work between CPUs and parser CPUs as indicated by **(parser)**.

Protocol Field	Linux ins. count	SDPU ins. count
EtherType with lookup	> 50	10 **2 (parser)**
IP protocol, IHL with length validation	30	**4 (parser)**
Addresses and ports	> 100 (mostly look up cost)	10 (lookups are asynchronous)
IPv4 header checksum	20	2
IP flags, Ident, Frag Offset	10	10
TOS	15	15
TCP flags	30	**2 (parser)**
TCP checksum	40 (incl. pullup)	10
Other TCP fields	30	30
TCP options	40	40 **4 (parser)**
Syn cookie creation	50	50
Total	>500	177 **12 (parser)**

[15] Unlike x86 and ARM, RISC-V lacks processor flags so there is no add-with-carry instruction typically used to make Internet checksum calculation efficient. We created "checksum add" and "checksum fold" RISC-V instructions that don't require add-with-carry.

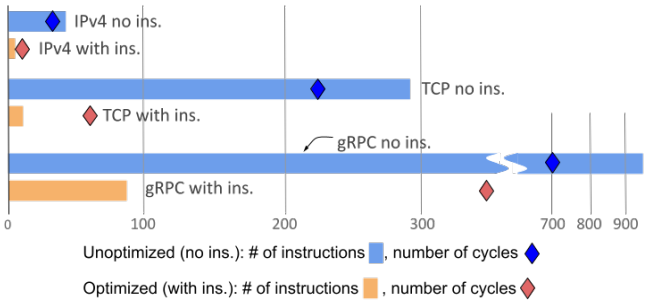

Figure 13: Parser performance. Test cases are IPv4, TCP with options, and a gRPC example with nested protobufs. For each case, the number of instructions and cycles are shown for an x86 implementation using plain instructions, and an implementation in RISC-V with SDPU parser instructions. Note that lower is better.

6.2.3 *Parser CPU Performance*

To optimize parsing performance, we implemented a set of parser instructions in RISC-V [49]. These instructions encompass the domain specific operations of parsing, and can efficiently parse a wide range of protocols including some of the more "difficult" protocol constructs to parse like Type Length Values (TLVs), flag-fields like in GRE [15], and protobufs [41] for gRPC[16] [20].

Each parser instruction replaces five to three hundred CPU integer instructions with equivalent functionality. For parsing Internet protocols, we expect the ratio of parser instructions to equivalent integer CPU instructions to be about 15:1. Parser instructions have a lower IPC which we estimate to be about 0.4 on average. Assuming a CPU IPC of 1.4 for parsing using integer instructions, the relative throughput speedup from parser instructions is 4.3.

Figure 13 compares parsing performance between the SDPU parser and x86 for different protocols. Figure 14 shows a detailed comparison for parsing the IPv4 header.

6.2.4 *Throughput analysis*

The components of the SDPU (Figure 12) run in parallel in a pipeline so throughput is bounded by the minimum throughput of any component in the processing path. CPUs are likely to be the throughput bottleneck since they are programmable and the biggest performance variable (non-programmable components can presumably be provisioned with sufficient resources for a desired load).

Per Table 1, a worker CPU requires 177 instructions to process a TCP SYN packet. Assuming an IPC of 1.4, that gives about 15.8 million pps on a single 2GHz CPU. So, sixty-four worker CPUs are needed to sustain 1 billion pps throughput.

Based on TCP SYN cookie processing in Table 1, twelve parser instructions are needed to parse a TCP SYN packet. Assuming an IPC of 0.4 for parser instructions that gives 66.6 million pps for a single 2GHz CPU. Hence, sixteen parser CPUs are needed to sustain 1 billion pps throughput.

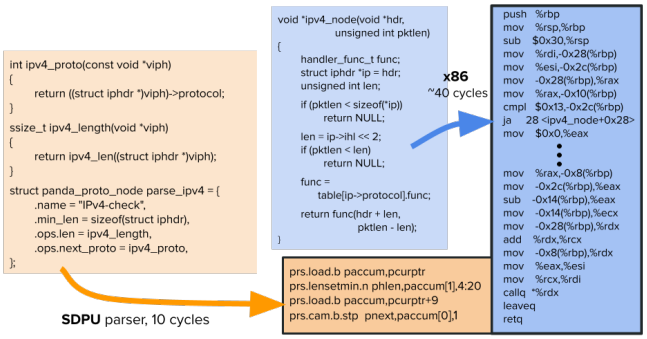

Figure 14: Parsing IPv4. The blue boxes show the C source code and disassembly for an x86 implementation, the orange boxes show the C source code and disassembly for the SDPU parser.

6.2.5 *Latency analysis*

Latency of a single packet through the SDPU is the sum of latencies of each of the inline processing components. Referring to Figure 12, a packet is processed in sequence by the dispatcher, cluster front end, parser, and cluster scheduler. The parser is user programmable and requires serialized processing per packet; referring to Table 1, twelve parser instructions are required to parse a SYN packet and with an IPC of 0.4 that gives a latency of thirty cycles. Extrapolating from the PoC and design, we expect a maximum latency of five cycles for each of the three uncore components. We budget five cycles of latency for the logic of sending and receiving packets. Summing these up, our estimate for packet processing latency of the parser and uncore components is fifty-five clock cycles.

We split processing for the SYN cookies program into five protocol layers: Ethernet, IPv4, TCP, TCP options, and SYN cookie creation. We assume there are enough threads for unconstrained vertical parallelism, so per packet processing latency is the maximum of that of any protocol layer (Equation 4). Per Table 2, the maximum processing latency is the SYN cookie creation with thirty-six cycles.

Adding the numbers, the per packet latency of the SDPU, assuming a 2GHz CPU and system clock, is about forty-five nanoseconds. As we mentioned, the SDPU is designed to minimize variance, so our expectation for 99.9th percentile tail latency is fifty nanoseconds.

Table 2: Instruction counts and cycles for the SDPU for protocol layer processing for TCP SYN cookies program. The instruction counts for each protocol layer are derived from Table 1. The number of cycles is calculated assuming an IPC of 1.4.

Protocol layer	Ins. count	Number of cycles
Ethernet	10	8
IPv4	37	27
TCP	40	29
TCP options	40	29
SYN Cookie creation	50	36

[16] In combination with integer instructions, parser instructions are Turing Complete [58] in that we can fall back to using plain integer instructions. We do this for parsing nested protobufs [41] in gRPC [20].

7. THE SINGLE FLOW BOTTLENECK

The benefits of parallelism are limited by the amount of parallelism available to programs, and by the overheads for orchestrating parallelism [23]. The effects in networking are particularly acute since networking is fundamentally serial data processing.

The worst case scenario in network processing is when all the packets being processed belong to a single flow that requires processing in a critical region. For instance, updating a TCP PCB is a *critical region* that must be serialized. If all packets being processed were for the same TCP connection, then a portion of processing for every packet is serialized with respect to other packets. In the case of multi-queue, this serialization is facilitated by steering packets of the same flow to a single CPU, however that results in no parallelism and maximum throughput is the reciprocal of the time it takes to process a packet (Figure 2(b)). In even a moderately high speed network, the time to process a packet is typically much greater than inter packet arrival time, hence throughput drops precipitously compared to idealized parallelism. We refer to this problem as the *Single Flow Bottleneck*, or *SFB.*

Addressing the Single Flow Bottleneck is challenging, even in hardware, since it's impossible to eliminate the need for serial execution in all workloads. Our strategy is to minimize the time spent in critical regions and to minimize the overheads of parallelism. Applying this strategy, we demonstrate a solution that addresses the Single Flow Bottleneck for two use cases: 1) a router forwarding packets [57], and 2) a high throughput TCP connection.

7.1 General strategy

In Figure 15(a), the solution to the Single Flow Bottleneck using horizontal parallelism is suggested. The portion of execution that must be serialized, for instance updating a TCP PCB, is identified as a *critical region*. Packets for a flow are processed by thread sets in horizontal parallelism and critical region processing is serialized and synchronized between threads. Processing outside the critical regions can be done in parallel. Throughput is bounded by the reciprocal of time processing the critical region plus synchronization overhead.

In Figure 15(b), the benefits of splitting critical regions are illustrated. Splitting critical regions is possible if there are no dependencies between the sub-regions. Throughput is bounded by the reciprocal of time processing the largest critical region plus synchronization overhead.

To minimize the time processing a critical region, its logic may be implemented in an accelerator. Such an accelerator could implement a very tight event loop in hardware to process requests quickly:

```
while (1) {
    if (!next_request)
       wait
    dequeue & process critical region
}
```

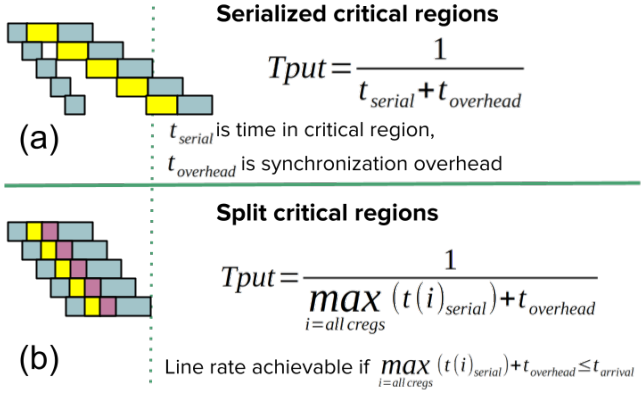

Figure 15: Addressing the single flow bottleneck. (a) applies horizontal parallelism with critical regions to process packets of the same flow on multiple threads for increased throughput. (b) shows how splitting critical regions increases throughput. The dotted line indicates line rate throughput, and critical regions are in yellow and purple. *Tput* gives the maximum throughput

For highest throughput, an accelerator should parallelize as much processing as possible outside of critical regions. This includes prefetching memory and staging requests for critical region processing.

If access to the critical region needs to be well ordered then an *ordered accelerator* can be used. An ordered accelerator associates a dependency with the accelerator so that waiting and resolving the dependency is handled transparently as part of accelerator processing. This eliminates the need for the program to use explicit wait and resolve primitives to synchronize accelerator processing.

7.2 Solving SFB in packet forwarding

In packet forwarding there are two operations performed in critical region processing: next hop lookup and packet transmission. Both of these can be implemented as ordered accelerators. The pseudo code for invoking them is:

```
next_hop = lookup_nexthop(dest_address)
send_packet(packet, next_hop)
```

Next hop lookup is usually a Longest Prefix Match [10], or LPM, lookup. The need for critical region processing arises from a requirement that routing must be consistent during a route change. The LPM lookup can be implemented as an ordered accelerator that is associated with a non-channel dependency. Internally, the accelerator can use a readers-writers lock [54] for the critical region, this allows route lookups to be done in parallel.

Packet transmission can be modeled as an accelerator that manages transmit queues with some queuing discipline. The critical region processing is an enqueue operation that can be efficiently implemented.

With optimized accelerator hardware, both the overhead of parallelism and critical region processing for route lookup and transmission accelerators can be minimized. Assuming critical region latency plus overhead is two nanoseconds, a 2Ghz clock frequency, and applying the equation in Figure 15(b), a forwarding rate of 1 billion pps is achievable.

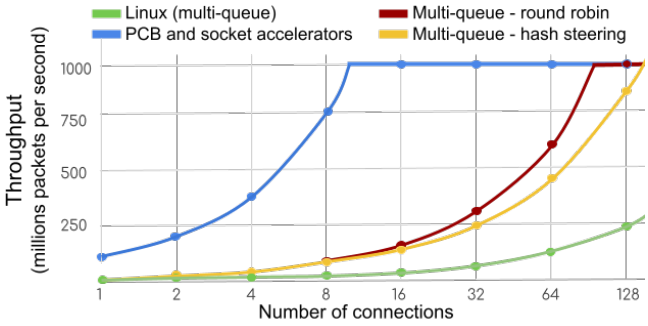

Figure 16: Maximum achievable throughput (pps) of TCP for different numbers of connections. Four methods are compared: Linux with multi-queue, SDPU with critical region accelerators, and optimized multi queue (with SDPU as a base) with hash and round robin steering. We assume that there are 128 threads (or queues) for each method where a thread can process 7.8 million packets per second for a maximum throughput of one billion pps.

7.3 Solving the SFB in TCP

We demonstrate addressing Single Flow Bottleneck in the TCP receive path, similar techniques would apply to the send path. There are two operations in the critical region of TCP receive processing: 1) reading and updating the PCB, and 2) enqueuing received data to the socket buffer. These operations are independent, so can we split them into two critical regions. We implement the processing of these in two accelerators: "update PCB" and "socket buffer". The "update PCB" accelerator will have the greater latency, so we'll focus on that one.

The "update PCB" accelerator can implement a fast path for TCP Header Prediction [6]. The core pseudo code[17] is:

```
if ((tcp_flags & FMASK) ==
  tp→pred_flags) && seq == tp→rcv_nxt) {
  if (len == 0)
     /* Pure ack handling */
  else if(ack == tp→snd_una) {
     tp→rcv_next += payload_len;
     if ((int)seq - tp→rcv_acked) >= 0)
        *need_ack = true
  }
}
```

In the fast path when the header is correctly predicted, critical processing is fully contained in the accelerator for lowest latency, and worker CPUs don't access the PCB so cache locality of flow related data structures isn't pertinent. For programmability, this logic could be implemented in a CPU thread that is driven by a hardware event loop. In a CPU the above logic requires about twenty instructions, and assuming an IPC of 1.4 and CPU clock frequency of 2 Ghz, the critical region processing latency is about eight nanoseconds. Adding two nanoseconds for synchronization overhead and applying the equation in Figure 15(b), the maximum throughput for a single TCP flow is 100 million pps. Figure 16 illustrates the benefits of this solution.

8. DISCUSSION & OPPORTUNITIES

In this paper we introduce an architecture for parallelism in network processing. Networking is inherently serialized processing, and our architecture elicits fine grained parallelism in serial data processing which has previously been considered an oxymoron. While network processing is the poster child of serial data processing, there are other use cases that could benefit from this architecture, including communications for collective operations [37], storage operations to a flash drive, or database queries.

A recent trend in network processing is programmability in the high performance path such as P4 [7]. For this, we advocate placing CPUs in the data path which we call *CPU-in-the-datapth*. This idea runs counter to conventional wisdom since performance of general purpose CPUs doesn't compare with that of specialized fixed function ASICs, however, in our architecture we employ *domain specific* CPUs for network processing. RISC-V [4], with its open ISA and ease of customization, is the perfect foundation for domain specific CPUs. We can take out things not needed for network processing, such as the floating point and vector unit, and we can add our own ISA extensions specific to network processing. We have defined new instructions in four classes: Parser, Threading and Synchronization (dependency primitives), Accelerator (general instruction to invoke accelerators), and Arithmetic (including instructions for checksum computation). With these extensions, horizontal and vertical parallelism, and in-line acceleration we project that CPU-in-the-datapath will achieve >85% of ASIC performance, but be fully programmable with lower cost and power consumption.

A programmable data path is a good start, however for widespread adoption it must be *easily programmable*. While Domain Specific Languages (DSLs), like P4 [42], have made inroads in data path programmability, they tend to be unfamiliar to programmers[18] and have a steep learning curve. With ease-of-use in mind, we eschew DSLs and provide a C library [39] for programming a data path. Our programming model is language agnostic, and we intend to support it in other languages including Python and Rust.

Compilers are essential to data path programmability. Their function is to take the programmer's expression of what they want to do and convert it into an optimized binary image for a target. Accordingly, we modified the LLVM CLANG [9] compiler specifically for our model. We have also defined a new Intermediate Representation (IR) in JSON called Common Parser Language (or CPL) [24] that encourages "freedom of front end languages". We are also investigating compiler techniques, including using CLANG and MLIR [35] as a parallelizing compiler [60], to find opportunities for both horizontal and vertical parallelism and automatically insert dependency primitives.

[17] This code snippet is adapted from Van Jacobson's famous "TCP in 30 Instructions" email [26]. The Linux implementation of TCP Header Prediction [30] "loosely follows" that with some additional processing of TCP timestamps and the TCP receive window.

[18] We note there are 15.7 million Python, & 11 million C/C++ programmers worldwide [50] and these languages are generally familiar.

REFERENCES

[1] About QEMU. *QEMU Documentation.* https://www.qemu.org/docs/master/about/index.html

[2] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. 2023. *Operating Systems: Three Easy Pieces* (Version 1.10). Arpaci-Dusseau Books. Chapter 30. https://pages.cs.wisc.edu/~remzi/OSTEP/

[3] Austin Appleby. Murmur3 hash code in MurmurHash3.cpp. https://github.com/aappleby/smhasher/blob/master/src/MurmurHash3.cpp

[4] Krste Asanović and David A. Patterson, Instruction Sets Should Be Free: The Case For RISC-V. April 2014. *Electrical Engineering and Computer Sciences University of California at Berkeley*. Technical Report No. UCB/EECS-2014-146. https://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-146.pdf

[5] Jean-Philippe Aumasson and Daniel J. Bernstein. September 2018. SipHash: a fast short-input PRF. In *Cryptology ePrint Archive*. https://eprint.iacr.org/2012/351.pdf

[6] D. Borman, B. Braden, V. Jacobson, and R. Scheffenberger (E.d). September 2014. *GTCP Extensions for High Performance*. RFC 7323. Section 5.6. DOI 10.17487/RFC7323. https://datatracker.ietf.org/doc/html/rfc7323

[7] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard. Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: programming protocol-independent packet processors. In *ACM SIGCOMM Computer Communication Review*. Volume 44 Issue 3. pp 87–95. DOI 10.1145/2656877. https://doi.org/10.1145/2656877.2656890

[8] Pong P. Chu. 2006. *RTL Hardware Design Using VHDL*. John Wiley & Sons, Inc., Hoboken, New Jersey. ISBN-13: 978-0-471-72092-8

[9] Clang: a C language family frontend for LLVM. https://clang.llvm.org/

[10] Douglas Comer. 2008. *Computer Networks and Internets* (5th edition). Pearson/Prentice Hall. p. 386. ISBN 978-0-13-606698-9

[11] J. Corbet. 2008. Improving TCP syncookies. *LWN.net.* https://lwn.net/Articles/277146/

[12] J. Corbet. 2008. Multiqueue Networking. *LWN.net*, https://lwn.net/Articles/289137/

[13] Willem de Brujin. 2013, rps: selective flow shedding during softnet overflow. *Linux kernel patch*. https://patchwork.ozlabs.org/project/netdev/patch/1366393612-16885-1-git-send-email-willemb@google.com/

[14] J. Dean and L. A. Barroso. The tail at scale. In *Communications of the ACM*, Volume 56 Issue 2. pp. 74–80. DOI 10.1145/2408776.2408794. https://www.barroso.org/publications/TheTailAtScale.pdf

[15] D. Farinacci, T. Li, S. Hanks, D. Meyer, and P. Traina. March 2000. *Generic Routing Encapsulation (GRE).* RFC 2784. RFC Editor. DOI 10.17487/RFC2784. https://datatracker.ietf.org/doc/html/rfc2784

[16] Michael J. Flynn. September 1972. Some Computer Organizations and Their Effectiveness. In *IEEE Transactions on Computers*. Volume C-21 Issue 9. pp. 948–960. DOI 10.1109/TC.1972.5009071

[17] Glen Gibb, George Varghese, Mark Horowitz, and Nick McKeown. October 2013. Design Principles for Packet Parsers. In *Architectures for Networking and Communications Systems*. DOI 10.1109/ANCS.2013.6665172. http://yuba.stanford.edu/~nickm/papers/ancs48-gibb.pdf

[18] R. Guerraoui, H. Guiroux, R. Lachaize, V. Quéma, and V. Trigonakis. July 1990. Lock–Unlock: Is That All? A Pragmatic Analysis of Locking in Software Systems. In *ACM Transactions on Computer Systems*. Volume 36 Issue 1 Article No.: 1. pp 1–149. https://doi.org/10.1145/3301501

[19] V. Gramoli. 2015. More than you ever wanted to know about synchronization: Synchrobench, measuring the impact of the synchronization on concurrent algorithms. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM. pp. 1–10. https://dl.acm.org/doi/10.1145/2858788.2688501

[20] gRPC: Core concepts, architecture and lifecycle. https://grpc.io/docs/what-is-grpc/core-concepts/

[21] John L. Hennessy and David A. Patterson. 2019. *Computer architecture: a quantitative approach* (6th edition). Morgan Kaufmann Publishers, an imprint of Elsevier. Cambridge, Mass. ISBN 978-0-12-811905-1

[22] John L. Hennessy and David A. Patterson. 2019. *Computer architecture: a quantitative approach* (6th edition). Morgan Kaufmann Publishers, an imprint of Elsevier. Cambridge, Mass. p. 540. ISBN 978-0-12-811905-1

[23] John L. Hennessy and David A. Patterson. 2012. *Computer architecture: a quantitative approach* (5th edition). Morgan Kaufmann Publishers, an imprint of Elsevier. Waltham, Mass. pp. 349-351. ISBN 978-0-12-383872-8

[24] Tom Herbert, Pratyush Khan, and Aravind Buduri. 2022. High Performance Programmable Parser. Slides from *Netdev 0x16* conference. Slides 9-17. https://netdevconf.info/0x16/papers/11/High%20Performance%20Programmable%20Parsers.pdf

[25] Tom Herbert and Willem de Brujin. May 2014. Scaling in the Linux Stack. *Linux Kernel Documentation.* https://docs.kernel.org/networking/scaling.html

[26] Van Jacobson. September 1993. Email to Craig Partridge. "*Fwd: TCP in 30 instructions*". https://www.pdl.cmu.edu/mailinglists/ips/mail/msg00133.html

[27] Bob Jenkins. November 2013. A hash function for hash Table lookup. http://www.burtleburtle.net/bob/hash/doobs.html

[28] Hugo Krawczyk. 1995. New Hash Functions for Message Authentication. In *EUROCRYPT '95. Lecture Notes in Computer Science*. Vol. 921. pp. 301–310. DOI 10.1007/3-540-49264-X_24. ISSN 0302-9743. https://link.springer.com/content/pdf/10.1007/3-540-49264-X_24.pdf

[29] Jorg Liebeherr and Erhan Yimaz. May 1999. Workconserving vs. Non-workconserving Packet Scheduling: An Issue Revisited. In *Proceedings of IEEE/IFIP IWQoS '99*. https://www.comm.utoronto.ca/~jorg/archive/papers/iwqos99-conserv.pdf

[30] Linux kernel *tcp_rcv_established* in tcp_input.c. https://github.com/torvalds/linux/blob/master/net/ipv4/tcp_input.c

[31] The LLVM Compiler Infrastructure. llvm.org

[32] Dave McCracken. June 2002. POSIX Threads and the Linux Kernel. In *Proceedings of the 2002 Ottawa Linux Symposium.* pp. 330-337. https://www.kernel.org/doc/ols/2002/ols2002-pages-330-337.pdf

[33] Durganshu Mishra. October 2023. The Power of Parallel Programming: Why It's a Necessity. *Medium.com*. https://medium.com/@durganshu/the-power-of-parallel-programming-why-its-a-necessity-2429e4622f5e

[34] Gordon E. Moore. April 1965. Cramming More Components onto Integrated Circuits. *Electronics.* pp. 114-117. Reprinted in *Proceedings of the IEEE*, Vol. 86, No.1., January 1998. https://www.cs.utexas.edu/~fussell/courses/cs352h/papers/moore.pdf

[35] Multi-Level Intermediate Representation Overview. https://mlir.llvm.org/

[36] Gor Nishanov. November 2018. Fibers under the magnifying glass. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1364r0.pdf

[37] Nvidia.Collective Operations. NCCL2.19. https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/usage/collectives.html

[38] Open Systems Interconnection — Basic Reference Model: The Basic Model. June 1999. *ISO/IEC 7498-1:1994.* https://www.iso.org/standard/20269.html

[39] PANDA (Protocol And Network Datapath Acceleration). https://github.com/panda-net/panda

[40] W. Petersen and D. T. Brown. January 1961. Cyclic Codes for Error Detection. In *Proceedings of the IRE*. Volume 49 Issue 1. pp. 228–235. DOI 10.1109/JRPROC.1961.287814. https://apt.cs.manchester.ac.uk/ftp/pub/apt/papers/Peterson-Brown_61.pdf

[41] Protocol Buffers Documentation. https://protobuf.dev/

[42] $P4_{16}$ Language Specification, version 1.2.2. May 2021. The P4 Language Consortium. https://p4.org/p4-spec/docs/P4-16-v1.2.2.html

[43] Vivek Ratan. Docker: A Favourite in the DevOps World. https://www.opensourceforu.com/2017/02/docker-favourite-devops-world/

[44] Thomas Rauber and Gudula Runger. 2013. *Parallel Programming for Multicore and Cluster Systems*. Springer. Chapter 3. ISBN 978-3-642-43806-6

[45] Thomas Rauber and Gudula Runger. 2013. *Parallel Programming for Multicore and Cluster Systems*. Springer. pp. 9-14. ISBN 978-3-642-43806-6

[46] Thomas Rauber and Gudula Runger. 2013. *Parallel Programming for Multicore and Cluster Systems*. Springer. pp. 154-155. ISBN 978-3-642-43806-6

[47] Thomas Rauber and Gudula Runger. 2013. *Parallel Programming for Multicore and Cluster Systems*. Springer. p. 157. ISBN 978-3-642-43806-

[48] David Rotman. February 2020. We're not prepared for the end of Moore's Law. *MIT Technology Review*. https://www.technologyreview.com/2020/02/24/905789/were-not-prepared-for-the-end-of-moores-law/

[49] SiPanda RISC-V Parser Instructions (not yet published)

[50] Size of programming language communities worldwide as of 2022. statista. https://www.statista.com/statistics/1241923/worldwide-software-developer-programming-language-communities/

[51] W. Richard Stevens and Gary R. Wright. 2017. *TCP/IP Illustrated. Vol. 1, The protocols*. Addison-Wesley. Chapter 22. ISBN 978-0-13-476013-1

[52] Andrew S. Tanenbaum and Herbert Bos. 2014. *Modern Operating Systems* (4^{th} ed.). Pearson. p. 153. ISBN-13: 978-0-13-359162-0

[53] Andrew S. Tanenbaum and Herbert Bos. 2014. *Modern Operating Systems* (4^{th} ed.). Pearson. p. 28. ISBN-13: 978-0-13-359162-0

[54] Andrew S. Tanenbaum and Herbert Bos. 2014. *Modern Operating Systems* (4^{th} ed.). Pearson. pp. 171-172. ISBN-13: 978-0-13-359162-0

[55] James T. Townsend. July 1990. Serial vs. Parallel Processing: Sometimes They Look Like Tweedledum and Tweedledee but They Can (and Should) Be Distinguished. In *Psychological Science.* Volume 1 No. 1. pp. 46-54. DOI 10.1111/j.1467-9280.1990.tb00067.x

[56] USPTO, Parallelism in Serial Pipeline Processing. *Patent pending*

[57] A. Toonk. March 2020, Linux Kernel and Measuring network throughput. https://atoonk.medium.com/linux-kernel-and-measuring-network-throughput-547c3b68c4d2

[58] A. M. Turing. July 1948. *Intelligent Machinery* (Report). National Physical Laboratory. pp. 3-4. https://weightagnostic.github.io/papers/turing1948.pdf

[59] A. Viviano. 2023. Introduction to Receive Side Scaling. Microsoft. https://learn.microsoft.com/en-us/windows-hardware/drivers/network/introduction-to-receive-side-scaling

[60] Michael Wolfe. 1996. Parallelizing Compilers. *Oregon Graduate Institute.* https://dl.acm.org/doi/pdf/10.1145/234313.234417#:~:text=A%20%E2%80%9Cparallelizing%20compiler%E2%80%9D%20is%20typically,array%20assignments%20or%20parallel%20loops.

[61] Xilinx. 2018. *Alveo U200 and U250 Data Center Accelerator Cards Data Sheet.* https://www.avnet.com/opasdata/d120001/medias/docus/190/XLX-A-U200-P64G-PQ-G-Datasheet.pdf

[62] J. Ziv and A. Lempel. 1978. Compression of individual sequences via variable-rate coding. In *IEEE Transactions on Information Theory*. Volume 24 Issue 5. pp 530-536. DOI 10.1109/TIT.1978.1055934. https://paginas.fe.up.pt/~sam/TI/artigos/ziv78compression.pdf